Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5540--99-8359

Tools for Simplifying Proofs of Properties of Timed Automata: The TAME Template, Theories, and Strategies

MYLA ARCHER

Center for High Assurance Computer Systems
Information Technology Division

March 26, 1999

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 26, 1999	

4. TITLE AND SUBTITLE

Tools for Simplifying Proofs of Properties of Timed Automata:
The TAME Template, Theories, and Strategies

5. FUNDING NUMBERS

PE - 62234N
WU - 55-3978-09

6. AUTHOR(S)

Myla Archer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5540--99-8359

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Ballston Centre Tower One, 800 North Quincy Street
Arlington, VA 22217-5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

A

13. ABSTRACT (*Maximum 200 words*)

The tool TAME is intended to provide automated support to simplify specifying and reasoning about timed automata using PVS. This report documents the current specification template, standard theories, and PVS strategies upon which TAME is based. The TAME system is now available for limited distribution by contracting the author.

14. SUBJECT TERMS

Formal methods
Timed automata
Automated theorem proving
Verification

15. NUMBER OF PAGES

56

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

CONTENTS

TOOLS FOR SIMPLIFYING PROOFS OF PROPERTIES OF TIMED AUTOMATA: THE TAME TEMPLATE, THEORIES, AND STRATEGIES

1. Introduction.

TAME is a high-level PVS interface that is intended to provide automated support to simplify specifying and reasoning about Lynch-Vaandrager (LV) timed automata [LV_91,HL_94] with PVS. TAME is based upon a specification template for timed automata, a set of standard theories, and a set of standard PVS strategies. There are two categories of TAME strategies: generic strategies and local strategies. Local strategies are generated from a template instantiation, and are therefore different in detail for each particular timed automaton. Similarly, there are two categories of standard TAME theories: the generic theories and local theories, where the local theories are generated from a given template instantiation.

TAME has been previously described in several papers, including [AH_96a], [AH_96b], [AH_97a], [AH_97b], [AHS_98], and [AH_98]. The last documents in detail the state of TAME at the time [AH_96a] was written and summarizes more recent developments. Perhaps the most significant of these developments is the design of a generic strategy AUTO_INDUCT (for "automaton induction") that does the major initial work in the induction proof of a state invariant. AUTO_INDUCT and several other TAME strategies that used not to be application-independent are so now, being in effect "parameterized" by the local strategies and local theories. TAME also now supports reasoning about I/O automata [LT_89] and SCR automata (see [AHS_98]). This report documents in detail the current state of TAME support for reasoning about LV timed automata and I/O automata.

Section 2 of this report documents the TAME strategies from the user's point of view. Appendix 1 contains the TAME timed automaton template. Appendix 2 contains the generic theories, with the exception of the theory **atexecs**, which supports reasoning about admissible timed executions of timed automata, and which is documented in [AH_98]. Appendix 3 contains the code of the current TAME strategies. Finally, Appendix 4 contains as an example the template instantiation, local theories, and local strategies for the Steam Boiler Controller studied in [AH_97a].

2. TAME Strategies for the TAME User.

(APPLY_GENERAL_PRECOND)

Effect: Causes the general precondition for the action of an induction branch to be expanded.

Use: Sometimes helpful for timed automata; irrelevant for untimed I/O automata.

(APPLY_IND_HYP <arg1> ... <arg*n* >)

Effect: Applies the inductive hypothesis to the arguments.

Use: Needed when a universally quantified formula is being proved by induction, and an instantiation of the inductive hypothesis other than the default is required in an induction branch of the proof. (The default instantiation is the list of skolem constants that is automatically generated for the inductive conclusion of the particular proof branch.)

(APPLY_INV_LEMMA <inv-name> [<state>] [<arg1> ... <arg*n* >])

Effect: Applies the invariant lemma lemma_<inv-name> to the state <state>, if present, and the argument list. If <state> is omitted, then the state is assumed to be the state named ''prestate'', which is the standard name for the prestate of the transition in the induction step of any induction proof, the initial state in the base case, and the generic state in a direct proof. The resulting formula or formulas are labeled ''lemma_<inv-name>''.

Use: Needed in proving invariant lemmas that would require strengthening to be inductive. Also usually needed in direct (non-induction) proofs.

Manuscript approved January 8, 1999.

(APPLY_LEMMA <lemma-name> [<arg1> ... <arg*n* >])

Effect: Applies the lemma <lemma-name> to the arguments. The resulting formula or formulas are labeled "<lemma-name>".

Use: Sometimes needed in applying lemmas about the data types involved in the automaton state to state variables.

(APPLY_SPECIFIC_PRECOND)

Effect: Causes the specific precondition for the action of an induction branch to be expanded, and, if the precondition is a conjunction, causes the conjuncts in the precondition to be separated and given secondary individual labels as "specific-precondition_part_1", ... , "specific-precondition_part_n", according to their position in the conjunction.

Use: Needed when the truth of the induction step corresponding to an action depends on the precondition of the action.

(AUTO_INDUCT)

Effect: Causes the proof of a formula of the form

(*) (*FORALL* (*s* :*states*):*reachable* (*s*) => *Inv_* <*invname* >(*s*))

to be split into branches corresponding to the base case and each action case of an induction proof. When Inv_<invname> expands to a universally quantified formula, automatically skolemizes the inductive conclusion, and instantiates the inductive hypothesis with the skolem constants.

Labels parts of the base case as "prestate-definition" and "conclusion"; when the conclusion is a disjunction, the disjuncts are separated, and given secondary labels as "conclusion_part_1", ... , "conclusion_part_n", in the order in which they appear in the disjunction. (Implications are treated as disjunctions.)

Labels parts of each inductive case as "prestate-reachable", "poststate-reachable", "general-precondition", "specific-precondition", "OKstate?", "inductive-hypothesis", and "inductive-conclusion". When the inductive hypothesis is a conjunction or the inductive conclusion is a disjunction, the parts are separated and, in analogy with "conclusion" in the base case, given secondary individual labels as "..._part1" through "..._part*n* ". (A formula *A* IFF *B* is treated as the conjunction *A* =>*B* AND *B* =>*A* .)

Each proof branch is tested to see if it can be proved with some simple propositional reasoning and the application of standard rewrites and decision procedures. If the proof fails, the branch is unaffected. As a result, only the "nontrivial" proof branches are returned to the user for further interactive proof.

Comments are printed at the beginning of each proof branch in the saved proof indicating either that it corresponds to the base case or that is the (induction) branch corresponding to some particular action.

The following naming conventions are followed: In the base case, the initial state is called "prestate". In every induction step, the prestate is called "prestate", and the poststate is called "poststate". When an action has parameters <param_1>, ... , <param_m>, these are replaced in the corresponding action case by the skolem constants "<param_1>_action", ... , "<param_m>_action". When the invariant is universally quantified over variables <name_1>, ... , <name_n>, the inductive conclusion is skolemized with the skolem constants "<name_1>_theorem", ... , "<name_n>_theorem", and the inductive hypothesis is instantiated

with these skolem constants. The strategy APPLY_IND_HYP can be used if a different instantiation of the inductive hypothesis is needed.

Use: Should be be applied only as the first step in the proof of a formula of the form (*). Is the appropriate first step in a proof by induction (as opposed to a proof that uses direct deduction from other invariant lemmas). Sometimes is the only step needed in an induction proof.

(CANCEL_FORMULAS)

Effect: Removes the hypothesis of an implication in the antecedent of a sequent or the first conjunct in a conjunction in the consequent of the sequent if the hypothesis or first conjunct can be trivially deduced from the sequent.

Use: This strategy was developed to compensate for the fact that the PVS rule ASSERT performs the desired operation only if the hypothesis or conjunct to be cancelled is a simple formula (e.g., not quantified, not a conjunct or disjunct, etc.). It does not correspond to any substantive step used in high level hand proofs. It is helpful for cleaning up the sequent visually and, in some cases, as a preparatory step to certain TAME strategies that employ forward chaining.

(COMPUTE_POSTSTATE [<label>])

Effect: Causes the definition of the variable "poststate" that is saved among the hidden formulas (by AUTO_INDUCT) to be substituted for "poststate" (only in formulas labeled <label>, if the argument <label> is present), and performs some computation on the resulting expression by applying the definition of the transition function and a certain amount of simplification.

Use: This strategy is useful, for example, when (using SUPPOSE) a case split based on some property of the poststate has occurred in an induction branch of an induction proof.

(CONST_FACTS)

Effect: Introduces as separate formulas the facts about the constants used in a specification that are formalized in the axiom `const_facts`, a standard template entry (whose default is **true**).

Use: Needed when the correctness of a lemma or proof branch depends on the known facts about the constants. These facts are typically in the form of inequalities or other relations between the constants.

(DIRECT_PROOF)

Effect: Performs the standard set of initial steps in the direct proof of a formula of the form

(*) (*FORALL* (*s* :*states*):*reachable* (*s*) => *Inv_* <*invname* >(*s*)).

Causes the formula to be skolemized and separated into parts labeled "prestate-reachable" and "conclusion". When Inv_<invname> expands to a quantified formula, automatically skolemizes the conclusion, and when the result is a disjunction, separates the disjuncts and gives them secondary labels as "conclusion_part_1", ... , "conclusion_part_n".

The following naming conventions are followed: The state being reasoned about is called "prestate". When the invariant is universally quantified over arguments <arg_1>, ... , <arg_n>, the conclusion is skolemized with the skolem constants "<arg_1>_theorem", ... , "<arg_n>_theorem".

Use: Should be be applied only as the first step in the proof of a formula of the form (*). Appropriate as the first step in the direct proof of an invariant lemma as the consequence of previously proved invariant lemmas.

(EPSILON_WITNESS <expr>)

Effect: This strategy instantiates the existence proof obligation (labeled "epsilon axiom existence proof obligation") generated by an application of the TAME strategy USE_EPSILON.

Use: This strategy is required on the companion branch generated by any application of the TAME strategy USE_EPSILON. When many applications of USE_EPSILON are made on the main proof branch, it may be most convenient to discharge this proof obligation using the command

(BRANCH (USE_EPSILON <eps_pred_name> [<eps_pred_args>])
((SKIP) (EPSILON_WITNESS <expr>)))

(possibly a future TAME strategy) to avoid confusing multiple use of the obligation label.

(FOCUS_ON <label>)

Effect: Repeatedly interleaves propositional simplification of all formulas with label <label> with the application of flattening and the PVS decision procedures to all formulas. Since descendants of formulas labeled <label> inherit this label, this strategy has the effect of completely simplifying the original set of formulas with label <label>. It may result in multiple proof branches.

Use: Often more efficient than TRY_SIMP (or PVS's GRIND) in completing the proof of a branch that "is now obvious". May be used when completing the "obvious" remaining part of a proof clearly depends on the case breakdown of in the set of formulas labeled <label>.

(INST_IN <label> <arg1> ... <arg*n* >)

Effect: Performs (FOCUS_ON <label>) followed by instantiation, if possible, of some formula labeled <label> with the values <arg1>, ... , <arg*n* >.

Use: This strategy is intended to partially make up for the fact that instantiation of internal parts of formulas is not supported in PVS. Although it sometimes leads to multiple proof branches, it often saves splitting a sequent in order to move the formula to be instantiated to the top level.

(NONVERBOSE)

Effect: Causes APPLY_INV_LEMMA, APPLY_LEMMA, APPLY_SPECIFIC_PRECOND, and CONST_FACTS to cease printing as comments the facts introduced into the proof. All new proofs are then nonverbose, until VERBOSE is invoked.

Use: Allows a more compact version of a proof to be simply generated. Can make editing a proof simpler.

(SKOLEM_IN <label> <name1> ... <name*n* >)

Effect: Performs "(FOCUS_ON <label>)" followed by skolemization, if possible, of some formula labeled <label> with the skolem constants <name1>, ... , <name*n* >.

Use: This strategy is intended to partially make up for the fact that skolemization of internal parts of formulas is not supported in PVS. Although it sometimes leads to multiple proof branches, it often saves splitting a sequent in order to move the formula to be skolemized to the top level.

(SPECIALIZE_INDUCTION_TO <name1> ... <name*n* >)

Effect: Skolemizes a formula labeled "inductive-conclusion" with the skolem constants <name1> ... <name*n* >, and instantiates an appropriate formula labeled "inductive-hypothesis" with these skolem constants. An alternate version, SPECIALIZE_INDUCTION_TO_2, uses SKOLEM_IN and INST_IN.

Use: Can be needed when a state formula being proved as an invariant has embedded quantifiers rather than a universal quantifier at the top level. For example:

*Inv_*<*invname* 1> = (*EXISTS* (*x*):*P* (*x*)) => *Q*
*Inv_*<*invname* 2> = *P* => (*FORALL* (*x*):*Q* (*x*))

Can also be useful when the state formula does have a universal quantifier at the top level, when there are embedded quantifiers as well.

(SUPPOSE <expr>)

Effect: Performs (CASE <expr>), and labels the formula <expr> with "Suppose" on the main proof branch (where it becomes an hypothesis) and with "Suppose not" on the companion proof branch (where its negation becomes an hypothesis). Inserts comments "Suppose [<expr>]" and "Suppose not [<expr>]" in the saved proof, at the tops of the appropriate proof branches.

Use: Exactly the same as for CASE (on one argument); the labels simply give more information on the role the formula <expr> is expected to play in the proof.

(TRY_SIMP)

Effect: Performs propositional simplification and application of the PVS decision procedures. Simplifies arithmetic operators and comparisons used with the type "time", which, since it includes infinity as a possible value, is represented as a DATATYPE in PVS. Applies standard simplifications associated with the other DATATYPEs in the specification, through a combination of rewrites and forward chaining.

Use: Normally invoked when the proof reaches the stage where a human would say "it is now obvious". Generally more efficient for this purpose than the PVS strategy GRIND. Its use as an intermediate proof step should be rare. However, when it is used as an intermediate step, it avoids destroying as much of the high-level representation of facts and data as GRIND destroys.

(USE_OKSTATE)

Effect: Expands the definition of "OKstate?", and computes the poststate that appears as an argument to "OKstate?" in the hypothesis of an induction proof branch.

Use: Needed in proofs associated with specifications in which the set of reachable states is limited by an invariant (which is given the standard name *OKstate?* in the TAME template).

(USE_EPSILON <pred_name> [<pred_arg1> ... <pred_arg*n* >])

Effect: Finds an instance, if there is one, of a use of the predicate *P* = <pred_name>(<pred_arg1>, ... , <pred_arg*n* >), and applies the epsilon axiom *epsilon_ax* to it. Automatically computes the domain type of *P*, so that it need not be specified to apply the epsilon axiom. If <pred_arg1> ... <pred_arg*n* > are not given, determines them from the expression in which <pred_name> is used. USE_EPSILON always generates a companion branch requiring an existence proof; EPSILON_WITNESS can be used to discharge the companion subgoal.

Use: Useful when nondeterminism in the effects of actions of a timed automaton is present and specified by means of the Hilbert ε. A typical example is a hybrid automaton in which the change in value of certain physical quantities due to time passage is described in terms of constraints rather than exact values, and the new value of such a quantity is expressed as ε(*P*) for some *P*. The arguments <pred_arg1> ... <pred_arg*n* > in *P* correspond to the elapsed time and other variables involved in the constraints. [Use of the Hilbert ε in the specification is safe *provided only state invariants of the automaton are being proved.* Because its value is deterministic (though

unspecified), it is not always safe for drawing conclusions about other specification properties (such as that a given sequence of actions starting from a given state always results in the same new state).]

(VERBOSE)

Effect: Causes APPLY_INV_LEMMA, APPLY_LEMMA, APPLY_SPECIFIC_PRECOND, and CONST_FACTS to resume printing as comments the facts introduced into the proof. All new proofs are then verbose, until NONVERBOSE is invoked.

Use: Causes most of the relevant information introduced in the course of a proof to be included in the proof script as comments.

3. References.

[AHS_98] M. Archer, C. Heitmeyer, and S. Sims, *TAME: A PVS Interface To Simplify Proofs for Automata Models*, Proc. User Interfaces for Theorem Provers 1998, Eindhoven University CS Technical Report (Eindhoven University of Technology, Eindhoven, Netherlands, July, 1998), pp. 42-49.

[AH_96a] M. Archer and C. Heitmeyer, *Mechanical Verification of Timed Automata: A Case Study*, Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96) (IEEE Computer Society Press, 1996), pp. 192-203.

[AH_96b] M. Archer and C. Heitmeyer, *TAME: A Specialized Specification and Verification System for Timed Automata*, Work-In-Progress Proc. 1996 IEEE Real-Time Systems Symp. (RTSS'96), pp. 3-6.

[AH_97a] M. Archer and C. Heitmeyer, *Verifying Hybrid Systems Modeled as Timed Automata: A Case Study*, Hybrid and Real-Time Systems (HART'97), Lect. Notes in Comp. Sci. **1201** (Springer-Verlag, 1997), pp.171-185.

[AH_97b] M. Archer and C. Heitmeyer, *Human-Style Theorem Proving Using PVS*, in E. L. Gunter and A. Felty, eds., Theorem Proving in Higher Order Logics (TPHOLs'97), Lect. Notes in Comp. Sci. **1275** (Springer-Verlag, 1997), pp. 33-48.

[AH_98] M. Archer and C. Heitmeyer, *Mechanical Verification of Timed Automata: A Case Studys*, Tech. Rept. NRL/MR/5546--98-8180, NRL, Washington, DC, 1998.

[HL_94] C. Heitmeyer and N. Lynch, *The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems*, Technical Report MIT/LCS/TM-51, Lab. for Comp. Sci., MIT, Cambridge, MA, 1994; also Technical Report 7619, NRL, Wash., DC 1994.

[LL_96a] G. Leeb and N. Lynch, *Proving Safety Properties of the Steam Boiler Controller: Formal Methods for Industrial Applications: A Case Study*, in: J.-R. Abrial, et al., eds., Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, Lect. Notes in Comp. Sci. **1165** (Springer-Verlag, 1996).

[LL_96b] G. Leeb and N. Lynch, *Correction Sheet* for *Proving Safety Properties of the Steam Boiler Controller*, at the web site http://theory.lcs.mit.edu/tds/papers/Leeb/corr_full.html.

[LT_89] N. Lynch and M. Tuttle, *An Introduction to Input/Output Automata*, CWI-Quarterly **2** (3) (Centrum voor Wiskunde en Informatica, Amsterdam, September, 1989), pp. 219-246.

[LV_91] N. Lynch and F. Vaandrager, *Forward and Backward Simulations for Timing-Based Systems*, Proc. of REX Workshop "Real-Time: Theory in Practice", Lecture Notes in Computer Science **600** (Springer-Verlag, 1991), pp. 397-446.

Appendix 1 : The TAME Timed Automaton Template.

Below is the template for a timed automaton specification in TAME. Arbitrary declarations permissible in a PVS theory may be added if desired, including declarations of constants, types, or axioms. The parts that *must* be filled in are labeled **<..*n*..>** for $n = 0,...,7$, and represent the following information:

<..0..> : any relationships among the constants in the specification—the default is **true**;

<..1..> : declarations of the non-time-passage actions of the automaton;

<..2..> : the type of the basic state (i.e., the non-time-related part of the state) of the automaton, which is usually a record type whose fields represent the state variables;

<..3..> : an arbitrary state predicate that can be used to restrict the state space—its default is **true**;

<..4..> : the preconditions for the non-time-passage actions;

<..5..> : the effects, if any, of time passage on the basic state variables;

<..6..> : the effects of the non-time-passage actions on state variables other than *now* ;

<..7..> : the remaining part of the initial condition on the state—actually, the form of the start state predicate **start** is flexible, but **start**(s) must imply $now(s) = zero$.

<timed-automaton name>: THEORY
BEGIN
IMPORTING time_thy
const_facts: AXIOM = **<..0..>** ;
actions : DATATYPE
BEGIN
nu(timeof:(fintime?)): nu?
<..1..>
END actions;
MMTstates: TYPE = **<..2..>**
IMPORTING states[actions,MMTstates,time,fintime?]
OKstate? (s:states): bool = **<..3..>** ;
enabled_general (a:actions, s:states):bool = now(s) >= first(s)(a) & now(s) <= last(s)(a);
enabled_specific (a:actions, s:states):bool =
CASES a OF
nu(delta_t): (delta_t > zero
& FORALL (a0: actions): NOT(nu?(a0)) => now(s) + delta_t <= last(a0,s)),
<..4..>
ENDCASES;
trans (a:actions, s:states):states =
CASES a OF
nu(delta_t): s WITH [now := now(s)+delta_t, **<..5..>**],
<..6..>
ENDCASES;
enabled (a:actions, s:states):bool =
enabled_general(a,s) & enabled_specific(a,s) & OKstate?(trans(a,s));
start (s:states):bool = (now(s) = zero) & **<..7..>** ;
IMPORTING machine[states, actions, enabled, trans, start]
END *<timed-automaton name>*

Appendix 2 : The TAME Generic Standard Theories.

The theory **machine** provides support for TAME's induction strategy AUTO_INDUCT, which applies the theorem `machine_induct`. The strategy AUTO_INDUCT also applies the lemma `reachable_trans` to introduce the reachability of the poststate into the hypotheses of every induction step in an induction proof. Having the reachability of the poststate among the hypotheses facilitates applying a state invariant lemma to the poststate in the proof of an induction step when necessary. The updated theory **machine** below is the extension of the theory **machine** in [AH_96a,AH_98] with `reachable_trans` and its supporting predicate definition `reachable_trans_fact`.

```
machine [ states, actions: TYPE,
          enabled: [actions,states -> bool],
          trans: [actions,states -> states],
          start: [states -> bool] ] : THEORY

BEGIN
 s,s1: VAR states
 a: VAR actions
 n,n1: VAR nat

 Inv: VAR pred[states];      % pred[states]  =  [states -> bool];

 reachable_hidden(s,n): RECURSIVE bool  =
    IF n = 0 THEN start(s)
    ELSE (EXISTS a, s1: reachable_hidden(s1,n - 1) & enabled(a,s1) & s = trans(a,s1))
    ENDIF
    MEASURE n

 reachable(s): bool  =  (EXISTS n: reachable_hidden(s,n))

 base(Inv) : bool  =  (FORALL s: start(s) => Inv(s))

 inductstep(Inv) : bool  =
    (FORALL s, a: reachable(s) & Inv(s) & enabled(a,s) => Inv(trans(a,s)))

 inductthm(Inv): bool  =  base(Inv) & inductstep(Inv) => (FORALL s: reachable(s) => Inv(s))

 machine_induct: THEOREM (FORALL Inv: inductthm(Inv))

 reachable_trans_fact(s,a) : bool  =  (reachable(s) & enabled(a,s) => reachable(trans(a,s)));

 reachable_trans: LEMMA (FORALL s,a : reachable_trans_fact(s,a));

END machine
```

The theory **states** below is identical to that given in [AH_96a,AH_98].

```
states [ tasks, MMTstates, time : TYPE, fin_pred : [time -> bool] ] : THEORY
 BEGIN
  states: TYPE = [#  basic: MMTstates,  now: (fin_pred),  first, last: [tasks -> time]  #]
 END states
```

The definitions in the datatype **time** and the theory **time_thy** are the same as those in [AH_96a,AH_98]. Several lemmas have been added to **time_thy**. Of these, the lemmas `fintime_unique`, `fintime_elim_1`, `fintime_elim_2`, and `fintime_dur` are used as rewrite rules by TAME in aid of automating reasoning about inequalities and arithmetic for the data type **time**.

```
time: DATATYPE
 BEGIN
  fintime(dur:{r:real|r>=0}): fintime?
  infinity: inftime?
 END time

time_thy: THEORY
 BEGIN
  IMPORTING time
  zero: time = fintime(0);
  <= (t1,t2:time):bool = IF fintime?(t1) & fintime?(t2) THEN dur(t1) <= dur(t2)
                         ELSE inftime?(t2) ENDIF;
  >= (t1,t2:time):bool = IF fintime?(t1) & fintime?(t2) THEN dur(t1) >= dur(t2)
                         ELSE inftime?(t1) ENDIF;
  < (t1,t2:time):bool =  IF fintime?(t1) & fintime?(t2) THEN dur(t1) < dur(t2)
                         ELSE NOT(inftime?(t1)) & inftime?(t2) ENDIF;
  > (t1,t2:time):bool =  IF fintime?(t1) & fintime?(t2) THEN dur(t1) > dur(t2)
                         ELSE NOT(inftime?(t2)) & inftime?(t1) ENDIF;
  + (t1,t2:time):time =  IF fintime?(t1) & fintime?(t2) THEN fintime(dur(t1) + dur(t2))
                         ELSE infinity ENDIF;
  % Note that the definition of "–" is a problem when dur(t1) < dur(t2).  Subtraction of
  % time values in TAME specifications is therefore strongly discouraged.  Instead,
  % specifications should be reformulated to use addition instead.
  – (t1:time, t2:(fintime?)):time =
                         IF fintime?(t1) & dur(t1) >= dur(t2) THEN fintime(dur(t1) – dur(t2))
                         ELSE infinity ENDIF;
  % The following lemmas are used by the automatic strategy TRY_SIMP in reasoning about
  % time values.
  fintime_unique: LEMMA (FORALL (z1,z2:{r:real|r>=0}):(fintime(z1) = fintime(z2)) => (z1 = z2));
  fintime_elim_1: LEMMA (FORALL (z:{r:real|r>=0}, t:(fintime?)): (fintime(z) = t) => (z = dur(t)));
  fintime_elim_2: LEMMA (FORALL (z:{r:real|r>=0}, t:(fintime?)): (t = fintime(z)) => (dur(t) = z));
  fintime_dur: LEMMA (FORALL (t:(fintime?)): fintime(dur(t)) = t);
  trans_order: LEMMA (FORALL (t1,t2,t3:time): t1 <= t2 & t2 <= t3 => t1 <= t3)
 END time_thy
```

The theory **real_thy** contains various lemmas about the real numbers that have proved useful when reasoning about nonlinear real arithmetic. These and additional lemmas are being investigated as a basis for improved support for automating such reasoning as much as possible. Reasoning about nonlinear real arithmetic often arises in the context of hybrid automata, as discussed in [AH_97a]. The theory **real_thy** presented here is an extension of that presented in [AH_97a].

```
real_thy: THEORY
 BEGIN
   nonnegreal:TYPE = {r:real | 0 <= r};
   nonposreal:TYPE = {r:real | 0 >= r};
 % The following lemma is built in to the PVS prelude.
 % posreal_mult_closed:LEMMA (FORALL (x,y:real): (x > 0 & y > 0) => x*y > 0);
   nonnegreal_mult_closed:LEMMA (FORALL (x,y:real): (x >= 0 & y >= 0) => x*y >= 0);
   greater_diff_positive:LEMMA (FORALL (x,y:real): (x > y) = (x - y > 0));
   greater_eq_diff_nonnegative:LEMMA (FORALL (x,y:real): (x >= y) = (x - y >= 0));
   greater_posmult_closed:LEMMA (FORALL (x,y,z:real): (x > 0 & y > z) => x*y > x*z);
   greater_eq_nonnegmult_closed:LEMMA (FORALL (x,y,z:real): (x >= 0 & y >= z) => x*y >= x*z);
   twice:LEMMA (FORALL (x:real): 2*x = x + x);
   sq(x:real):real = x*x;
   diff_of_sq:LEMMA (FORALL (x,y:nonnegreal): (x > y) => (sq(x) > sq(y)));
   sq_nonneg:LEMMA (FORALL (x:real): sq(x) >= 0);
   nonpos_neg_quotient:LEMMA (FORALL (x:real,y:real): (x <= 0 & y < 0) => x/y >= 0);
   nonneg_pos_quotient:LEMMA (FORALL (x:real,y:real): (x >= 0 & y > 0) => x/y >= 0);
 END real_thy
```

Appendix 3 : Code for the TAME Strategies.

```
; ***                          Section 0                                ***
;
; *** Definitions of access and analysis functions for strategies. ***

;; The global variable *timed-auto-simp-strat* is set to |<timed-auto>_simp|
;; once the name <timed-auto> is determined in either auto_induct or
;; direct_proof.

(setq *timed-auto-simp-strat* 'skip)
(setq *timed-auto-forward-strat* 'skip)
(setq *timed-auto-verbose-proofs* t)
(setq *branch-counter* 0)

(defun has_quant (form)
   (let ((has-quant nil))
   (mapobject #'(lambda (x) (if has-quant t
                                (when (or (typep x 'forall-expr)
                                          (typep x 'exists-expr))
                                      (setq has-quant t) t)))
              form)
   has-quant))

(defun has_specific_precond_form (form)
   (let ((has-specific nil))
   (mapobject #'(lambda (x)
                  (if has-specific t
                      (when (and (application? x)
                                 (name? (operator x))
                                 (tc-eq (id (operator x))
                                        '|enabled_specific| ))
                      (setq has-specific t) t)))
              form)
   has-specific))

(defun has_specific_precond(forms)
  (eval (cons 'or
              (loop for x in forms collect (has_specific_precond_form x)))))

(defun is_state_var (x)
  (cond (x (string-equal (princ-to-string
                          (setq *type-comment* (type (typecheck
                           (setq *pc-parse-comment* (pc-parse x 'expr))))))
                         "states[actions, MMTstates, time, fintime?]"))))

(defun gather_labels (lis pred)
   (loop for x in lis when (apply pred (list x)) collect (label x)))

(defun gather_negsforms (sforms)
```

```
   (loop for x in sforms
         when (if (and (application? (formula x))
                       (name? (operator (formula x))))
                  (tc-eq (id (operator (formula x))) 'NOT)
                  nil)
         collect x))

(defun gather_possforms (sforms)
   (loop for x in sforms
         when (not (if (and (application? (formula x))
                            (name? (operator (formula x))))
                       (tc-eq (id (operator (formula x))) 'NOT)
                       nil))
         collect x))

(defun gather_fnums_label (sforms lab)
   (let ((negsforms (setq *negsforms-comment* (gather_negsforms sforms)))
         (possforms (setq *possforms-comment* (gather_possforms sforms))))
   (let ((negfnums (setq *negfnums-comment*
           (let ((fnum 0))
           (loop for x in negsforms do (setq fnum (- fnum 1))
                 when (member lab (label x)) collect fnum))))
         (posfnums (setq *posfnums-comment*
           (let ((fnum 0))
           (loop for x in possforms do (setq fnum (+ fnum 1))
                 when (member lab (label x)) collect fnum)))))
    (append negfnums posfnums))))

(defun gather_forms_label (sforms lab)
    (loop for x in sforms when (member lab (label x)) collect x))

; The function get_sform gets the sform in the list sforms whose formula number
; is fnum.

(defun get_sform (fnum sforms)
   (let ((possforms (gather_possforms sforms))
         (negsforms (gather_negsforms sforms)))
   (cond ((< fnum 0) (nth (- (- fnum) 1) negsforms))
         ((> fnum 0) (nth (- fnum 1) possforms)))))

; The function flatten_length computes the number of new formulae that will
; appear in the sequent if "flatten" is applied to formula whose formula number
; in sforms is fnum.

(defun flatten_length (fnum sforms)
   (let ((sform (setq *form-fl-len-comment* (get_sform fnum sforms))))
   (cond ((< fnum 0)
          (flatten_length_form fnum (argument (formula sform))))
         (t (flatten_length_form fnum (formula sform))))))

; The function flatten_length_form computes the number of new formulae that
```

```
; will appear in the sequent if "flatten" is applied to the "apparent" formula
; form whose formula number is fnum.  Only the sign of fnum matters.

(defun flatten_length_form (fnum form)
   (cond ((< fnum 0)
          (cond ((conjunction? form)
                 (+ (flatten_length_form -1 (args1 form))
                    (flatten_length_form -1 (args2 form))))
                ((iff? form) 2)
                (t 1)))
         (t
          (cond ((disjunction? form)
                 (+ (flatten_length_form +1 (args1 form))
                    (flatten_length_form +1 (args2 form))))
                ((implication? form)
                 (+ (flatten_length_form -1 (args1 form))
                    (flatten_length_form +1 (args2 form))))
                (t 1)))))

; The function get_form_label gets the first sform whose label is lab.

(defun get_form_label (sforms lab)
   (let ((fnums-label (gather_fnums_label sforms lab)))
   (cond (fnums-label (get_sform (car fnums-label) sforms))
         (t nil))))

(defun grab_trans (sform)
  (let ((trans_expr nil)
        (expr (formula sform)))
  (mapobject #'(lambda (x) (if trans_expr t
                               (when (and (typep x 'application)
                                          (tc-eq (id (operator x)) '|trans|))
                                     (setq trans_expr x)
                                     t)))
             expr)
   trans_expr))

(defun grab_reachable_expr (sform)
  (let ((reach_expr nil)
        (expr (formula sform)))
  (mapobject #'(lambda (x) (if reach_expr t
                               (when (and (typep x 'application)
                                          (tc-eq (id (operator x)) '|reachable|))
                                     (setq reach_expr x)
                                     t)))
             expr)
   reach_expr))

(defun grab_actions_ref (sforms)
   (cadr (actuals (module-instance
        (car (resolutions (operator (grab_reachable_expr (car sforms)))))))))
```

```
(defun grab_adt_constructors (sforms)
   (constructors (adt (type (car (resolutions (type-value
        (grab_actions_ref sforms)))))))))

(defun grab_thy_name_base (sforms)
   (id (module-instance (car (resolutions (type-value
        (grab_actions_ref sforms)))))))

(defun grab_inv (sforms)
   (operator (cadr (exprs (argument (expression (formula (car sforms))))))))

(defun grab_inv_def (sforms)
   (definition (declaration (car (resolutions (grab_inv sforms))))))

(defun insert_in_list (val lis) (if (member val lis) lis (cons val lis)))

(defun grab_list_types (sforms)
  (let ((list_types nil))
  (mapobject
   #'(lambda (x)
     (when (and (or (typep x 'type-name) (typep x 'adt-type-name))
                (eq (id x) '|list|))
           (setq list_types
                 (insert_in_list (print-string (car (actuals x))) list_types))
           t))
   sforms)
  list_types))

(defun get_ta_name (thyname)
   (let* ((extnchar (coerce '_ 'character))
          (endpt (position extnchar thyname :from-end t)))
   (subseq thyname 0 endpt)))

(defun simp_strat_name (ta_name)
   (intern (concatenate 'string ta_name "_simp")))

(defun forward_strat_name (ta_name)
   (intern (concatenate 'string ta_name "_forward")))

(defun unique_aux_name (ta_name)
   (concatenate 'string ta_name "_unique_aux"))

(defun rewrite_thy_1_name (ta_name)
   (concatenate 'string ta_name "_rewrite_aux_1"))

(defun rewrite_thy_2_name (ta_name)
   (concatenate 'string ta_name "_rewrite_aux_2"))

(defun action_skolem_name (argname)
   (concatenate 'string (string argname) "_action"))
```

```
(defun simple_induct_strat_step (inv invform opterm simp_strat)
   (let* ((opname (id opterm))
          (argnames (mapcar #'id (arguments opterm)))
          (arg_skolem_names (mapcar #'action_skolem_name argnames))
          (argsrest
            (let ((comma-list ""))
                 (loop for argname in (cdr arg_skolem_names) do
                       (setq comma-list
                         (concatenate 'string comma-list
                                      (format nil "~a~a" ", " argname))))
                 comma-list)))
   `(then ,(cons 'reduce_case (cons inv arg_skolem_names))
          (|match_univ_and_simp_probe| ,invform ,simp_strat)
          (set_up_poststate)
          (comment
           ,(if (not (string-equal argsrest ""))
                (format nil "~a~a~a~a~a~a" "Case " opname "("
                        (car arg_skolem_names) argsrest ")" )
                (if argnames (format nil "~a~a~a~a~a" "Case " opname "("
                                     (car arg_skolem_names) ")" )
                             (format nil "~a~a" "Case " opname))))
          (postpone))))

(defun mk_induction_strategy (sforms inv)
   (let ((thy_name (grab_thy_name_base sforms))
         (opterms (setq *opterms-comment* (grab_adt_constructors sforms)))
         (invform (setq *invform-comment* (grab_inv_def sforms))))
   (let ((list_types (setq *list-types-comment* (grab_list_types sforms))))
   (let ((list_rewrites_cmd (setq *list-rewrites-comment*
           (cons 'then
                 (mapcar
                   #'(lambda (x)
                     (list 'auto-rewrite-theory
                           (format nil "~a~a~a" "list_rewrites[" x "]")))
              (grab_list_types sforms))))))
   (let ((ta_name (get_ta_name (string thy_name))))
   (let ((simp_strat (setq *timed-auto-simp-strat*
          (simp_strat_name ta_name)))
         (forward_strat (setq *timed-auto-forward-strat*
          (forward_strat_name ta_name)))
         (unique_aux (setq *timed-auto-unique-aux*
          (unique_aux_name ta_name)))
         (rewrite_thy_1 (setq *timed-auto-rewrite-thy-1*
          (rewrite_thy_1_name ta_name)))
         (rewrite_thy_2 (setq *timed-auto-rewrite-thy-2*
          (rewrite_thy_2_name ta_name))))
   (let ((induction_branches (setq *ind-branch-comment*
           (loop for opterm in opterms collect
            (simple_induct_strat_step inv invform opterm simp_strat)))))
   `(then (auto-rewrite "fintime_unique" "fintime_elim_1" "fintime_elim_2"
                        "fintime_dur")
```

```
          (auto-rewrite-theory "bool_rewrites")
          ,list_rewrites_cmd
          (auto-rewrite-theory ,rewrite_thy_1)
          (branch (auto_cases ,inv)
                  ((then (base_case ,inv)
                         (replace "start-state")
                         (simplify)
                         (,simp_strat) (prop_probe)
                         (comment "Base case") (postpone))
                   (branch (induct_cases ,inv) ,induction_branches))))))))))


; ***                          Section 1                                  ***
;
; *** Strategies that support induction and direct proof of invariants. ***

(defstep auto_cases (inv)
  (let ((dummy (setq *capture-context-name* *current-context*)))
  (then (lemma "reachable_trans")
        (expand "reachable_trans_fact")
        (lemma "machine_induct")
        (expand "inductthm")
        (inst -1 inv)
        (split)))
  ""
  "Splitting into machine base and induction cases")

(defstep base_case (inv)
  (let ((sforms (setq *sforms-comment* (s-forms (current-goal *ps*))))
        (transexpr (setq *trans-comment* (grab_trans (cadr sforms))))
        (constrs (setq *constrs-comment*
         (constructors (adt (type (car (exprs (argument transexpr)))))))))
  (then (delete 2)
        (delete -1)                          ;special for "experimental"
        (expand "base")
        (skolem 1 "prestate")
        (with-labels (flatten-disjunct :depth 1)
                     (("start-state" "conclusion")))
        (flatten_labelled_formula "conclusion")
        (expand "start")
        (expand inv)))
  ""
  "Simplifying the machine base case")

(defstep induct_cases (inv)
  (let ((x (format nil "~a~a~a~a~a"
           "(LAMBDA (a: actions): (FORALL (s: states): reachable(s) & "
           inv
           "(s) & enabled(a,s) & reachable(trans(a,s)) => "
           inv
           "(trans(a,s))))")) )
```

```
  (then* (delete 2)
         (expand "inductstep")
         (lemma "actions_induction")
         (inst -1 x)
         (beta)
         (branch (split)
                 ((then (skolem 1 ("s_1" "a_1"))(inst -1 "a_1")(inst -1 "s_1")
                        (inst -2 "s_1" "a_1") (prop))
                  (skip)))))
  ""
  "Splitting the induction case on action class")

(defstep reduce_case (inv &rest vars)
  (let ((dummy1 (setq *varname-comment* (pc-parse "a_1" 'name)))
        (dummy2 (setq *varfind-comment*
                      (find *varname-comment* (collect-skolem-constants))))
        (sforms (setq *sforms-comment-red* (s-forms (current-goal *ps*))))
        (dummy3 (setq *trans-comment* (grab_trans (car sforms))))
        (cmd1 (cond (vars `(then (skolem 1 ,vars) (skolem 1 ("prestate"))))
                    (t '(skolem 1 "prestate")))))
  (let ((cmd `(then ,cmd1
                    (with-labels (flatten)
                        (("pre-state-reachable" "inductive-hypothesis"
                          "full-precondition" "post-state-reachable"
                          "inductive-conclusion")))
                    ,(cons 'reduce_case_2 (cons inv vars)))))
  (then (delete -1 2)
        cmd)))
  ""
  "Applying the standard simplification")

(defstep reduce_case_2 (inv &rest vars)
  (let ((dummy1 (setq *reduce2-sforms-comment* (s-forms (current-goal *ps*)))))
   (then
    (with-labels (then (expand "enabled") (flatten "full-precondition"))
                 (("general-precondition" "specific-precondition"
                   "OKstate?-precondition")))
    (expand "trans" "inductive-conclusion" :assert? NONE)
    (expand inv :assert? NONE)
    (simplify)))
  "" "")

(defstep auto_induct ()
  (let ((sforms (setq *sforms-comment* (s-forms (current-goal *ps*))))
        (inv (setq *inv-comment* (string (id (grab_inv sforms)))))
        (cmd (setq *cmd-comment* (mk_induction_strategy sforms inv))))
  cmd)
  ""
  "Taking care of the standard steps in the induction proof")

; The strategy set_up_poststate replaces each "trans" form of the post-state
```

```
; by the variable "poststate", which is defined via "name" to equal the "trans"
; form.  The definition of "poststate" is then hidden under the label
; "poststate-definition", to be retrieved as needed.

(defstep set_up_poststate ()
  (let ((sforms (setq *sforms-comment-2* (s-forms (current-goal *ps*))))
        (dummy (setq *poststate-comments-list* nil))
        (post (setq *post-comment*
          (gather_forms_label sforms '|post-state-reachable| )))
        (post-form-nums (setq *post-form-num-comment*
          (gather_fnums_label sforms '|post-state-reachable| ))))
  (let ((poststate
         (cond
          (post
           (setq *poststate-comments-list*
             (cons (print-string (argument (argument (formula (car post)))))
                   *poststate-comments-list*))
           (setq *poststate-comment*
                 (print-string (argument (argument (formula (car post))))))))
          (t nil)))
; Note that post-form-num is computed because "replace" refuses to replace to
; a label.  Hopefully that will change, since this computation depends on
; the "name" command adding a new formula at position -1.
        (new-post-form-num (setq *new-post-form-num-comment*
         (cond (post-form-nums (- (car post-form-nums) 1)) (t nil)))))
  (let ((dummy (setq *hereiam* t))
        (cmd (setq *auto-induct-2-cmd-comment*
          (cond
           (poststate '(then (with-labels (name "poststate" ,poststate)
                                          (("poststate-definition")))
                             (replace "poststate-definition" ,new-post-form-num)
                             (hide "poststate-definition")))
           (t '(skip))))))
  cmd)))
  ""
  "Taking care of the standard steps in the induction proof")

;; The strategy prop_probe is used to test whether the remainder of a proof
;; is "trivial".  It is part of several other "_probe" strategies.

(defstep prop_probe ()
  (then* (lift-if)
         (prop)
         (assert)
         (fail))
  "" "")

(defstep |match_univ_and_simp_probe| (invform simp_strat)
  (let ((dummy (setq *branch-counter* (+ 1 *branch-counter*)))
        (quantvars (setq *quantvars-comment*
         (if (forall-expr? invform) (bindings invform) nil))))
```

```
  (let ((skolemvars (setq *skolemvars-comment*
         (mapcar #'(lambda (x) (format nil "~a~a" x "_theorem"))
          (mapcar #'id quantvars)))))
  (let ((skolemcmd (setq *skolemcmd-comment*
         (if skolemvars `(skolem 1 ,skolemvars) '(skip))))
        (instcmd (setq *instcmd-comment*
         (if skolemvars (cons 'inst (cons -2 skolemvars)) '(skip))))
        (simp_stratcmd `(,simp_strat)))
  (then skolemcmd instcmd
        (flatten_labelled_formula '|inductive-hypothesis|)
        (flatten_labelled_formula '|inductive-conclusion|)
        simp_stratcmd (prop_probe)))))
  "" "")

(defstep direct_proof ()
  (let ((sforms (setq *sforms-comment* (s-forms (current-goal *ps*))))
        (inv-name (setq *inv-comment* (string (id (grab_inv sforms)))))
        (thy_name (grab_thy_name_base sforms)))
  (let ((ta_name (get_ta_name (string thy_name))))
  (let ((simp_strat (setq *timed-auto-simp-strat*
         (simp_strat_name ta_name)))
        (forward_strat (setq *timed-auto-forward-strat*
          (forward_strat_name ta_name)))
        (unique_aux (setq *timed-auto-unique-aux*
         (unique_aux_name ta_name)))
        (rewrite_thy_1 (setq *timed-auto-rewrite-thy-1*
         (rewrite_thy_1_name ta_name)))
        (rewrite_thy_2 (setq *timed-auto-rewrite-thy-2*
         (rewrite_thy_2_name ta_name))))
  (then (skolem 1 "prestate")
        (expand inv-name)
        (with-labels (flatten-disjunct :depth 1)
                     (("prestate-reachable" "conclusion")))
        (flatten_labelled_formula "conclusion")
        (direct_proof_2)))))
  ""
  "Doing the standard steps of a non-induction proof")

(defstep direct_proof_2 ()
  (let ((sforms (s-forms (current-goal *ps*)))
        (inv (setq *direct-inv-commnet*
         (formula (car (select-seq sforms 1)))))
        (simpcmd `(,*timed-auto-simp-strat*))
        (skolemcmd
         (if (forall-expr? inv)
             (let ((quantvars (setq *quantvars-comment* (bindings inv))))
             (let ((skolemvars (setq *skolemvars-comment*
                    (mapcar #'(lambda (x) (format nil "~a~a" x "_theorem"))
                     (mapcar #'id quantvars)))))
             `(skolem 1 ,skolemvars)))
             '(skip))))
```

```
  (then skolemcmd (flatten_labelled_formula '|conclusion|) simpcmd))
  ""
  "Doing the standard steps of a non-induction proof")

(defstep direct_induction ()
  (let ((sforms (setq *sforms-comment* (s-forms (current-goal *ps*))))
        (inv-name (setq *inv-comment* (string (id (grab_inv sforms)))))
        (thy_name (grab_thy_name_base sforms)))
  (let ((ta_name (get_ta_name (string thy_name))))
  (let ((simp_strat (setq *timed-auto-simp-strat*
         (simp_strat_name ta_name)))
        (forward_strat (setq *timed-auto-forward-strat*
         (forward_strat_name ta_name)))
        (unique_aux (setq *timed-auto-unique-aux*
         (unique_aux_name ta_name)))
        (rewrite_thy_1 (setq *timed-auto-rewrite-thy-1*
         (rewrite_thy_1_name ta_name)))
        (rewrite_thy_2 (setq *timed-auto-rewrite-thy-2*
         (rewrite_thy_2_name ta_name))))
  (then (skolem 1 "prestate")
        (expand inv-name)
        (with-labels (flatten-disjunct :depth 1)
                     (("prestate-reachable" "conclusion")))
        (flatten_labelled_formula "conclusion")
        (direct_induction_2)
        ))))
  ""
  "Doing the standard steps of a direct-induction proof")

(defstep direct_induction_2 ()
  (let ((sforms (s-forms (current-goal *ps*)))
        (inv (setq *direct-inv-comment*
          (formula (car (select-seq sforms 1)))))
        (ind-var-name (setq *direct-inductvar-comment*
          (id (car (bindings inv)))))
        (ind-skolem-var-name (format nil "~a~a" ind-var-name "_induct"))
        (ind-type-name (setq *direct-inducttype-comment*
          (let ((induct-type (car (types (car (bindings inv))))))
          (cond ((subtype? induct-type) (id (print-type induct-type)))
                ((adt-type-name? induct-type)
                 (id (type (car (resolutions induct-type)))))))))
        (ind-lemma-name (format nil "~a~a" ind-type-name "_induction"))
        (ind-inst-cmd (setq *direct-instcmd-comment*
          (format nil "~a~a~a~a~a" "(LAMBDA " (bindings inv) ": "
                  (expression inv) ")")))
        (simpcmd `(,*timed-auto-simp-strat*))
        (quantvars (setq *quantvars-comment0* (bindings (expression inv))))
        (skolemvars (setq *skolemvars-comment0*
          (mapcar #'(lambda (x) (format nil "~a~a" x "_theorem"))
                  (mapcar #'id quantvars))))
        (specializecmd
```

```
          (setq *direct-specialize-comment*
           (if (forall-expr? (expression inv))
               `,(cons 'specialize_induction_to skolemvars)
               '(skip))))
        (skolemcmd
          (if (forall-expr? (expression inv))
              `(skolem 1 ,skolemvars)
              '(skip)))
        (inductcmd
          (if (forall-expr? inv)
              (let ((inductvars (setq *inductvars-comment* (bindings inv))))
              (let ((inductname (setq *inductname-comment*
                    (id (car inductvars)))))
              `(then (apply_lemma_no_comment ,ind-lemma-name)
                     (inst -1 ,ind-inst-cmd)
                     (beta)
                     (branch
                      (split)
                      (,simpcmd
                       (then (hide "conclusion")
                             (label "induction-base-case"
                                    ,(format nil "~a" ind-lemma-name))
                             ,skolemcmd
                             (flatten_labelled_formula "induction-base-case")
                             ,simpcmd)
                       (then (hide "conclusion")
                             (label "induction-step-case"
                                    ,(format nil "~a" ind-lemma-name))
                             (skolem "induction-step-case"
                                     ,ind-skolem-var-name)
                             (with-labels
                              (flatten-disjunct "induction-step-case" :depth 1)
                              (("inductive-hypothesis" "inductive-conclusion")))
                             ,specializecmd
                             (flatten_labelled_formula "inductive-conclusion")
                             (flatten_labelled_formula "inductive-hypothesis")
                             ,simpcmd))))))
              '(skip))))
    inductcmd)
    ""
    "Doing the standard steps of a direct-induction proof")

; ***                          Section 2                                ***
;
; ***   Specialized simplification strategies for timed automata.  ***

; Simplification strategies that handle time definitions and other simple
; types of reasoning needed for timed automata.

(defstep time_etc_simp ()
  (then* (lift-if)
```

```
          (assert)
          (prop)
          (assert)
          (expand "time_thy.zero" :assert? NONE)
          (expand "time_thy.<=" :assert? NONE)
          (lift-if)
          (expand "time_thy.>=" :assert? NONE)
          (lift-if)
          (expand "<" :assert? NONE)
          (lift-if)
          (expand ">" :assert? NONE)
          (lift-if)
          (expand "+" :assert? NONE)
          (lift-if)
          (expand "-" :assert? NONE)
          (lift-if)
          (repeat* (then* (assert) (split) (lift-if) (flatten)))
          (repeat* (forward-chain "fintime_elim_1"))
          (repeat* (forward-chain "fintime_elim_2"))
          (list_forward_chain)
          )
  ""
  "Doing time-arithmetic")

(defstep list_forward_chain ()
  (then (then (forward-chain "listforward_6") (simplify))
        (then (forward-chain "listforward_5") (simplify))
        (forward-chain "listforward_7")
        (then (forward-chain "listforward_4") (simplify))
        (forward-chain "listforward_8")
        (forward-chain "listforward_1a")
        (forward-chain "listforward_1")
        (forward-chain "listforward_2")
        (forward-chain "listforward_3")
        (forward-chain "listforward_9"))
  "" "")

(defstep list_forward_chain_6 ()
  (branch (then (forward-chain "listforward_6")(simplify))
          ((list_forward_chain_6) (list_forward_chain_6) (skip)))
  "" "")

(defstep time_vals_simp ()
  (then* (expand "time_thy.zero")
         (expand "time_thy.<=")
         (expand "time_thy.>=")
         (expand "<")
         (expand ">")
         (expand "+")
         (expand "-")
         (lift-if)
```

```
        )
  ""
  "Doing time-arithmetic")

; The following shorter version of time_etc_simp was provided by Shankar
; at SRI.  It is equivalent in power to time_etc_simp, but testing has shown
; that while it is sometimes equally fast, it is sometimes several seconds
; slower.

(defstep time_etc_simp_shankar ()
  (then (stop-rewrite)
        (auto-rewrite-theory "time_thy")
        (repeat* (then (lift-if)(ground))))
  ""
  "Doing time-arithmetic")

; try_simp_new5 is included for compatiblity with numerous old proofs

(defstep try_simp_new5 () (try_simp) "" "")

(defstep try_simp ()
   (let ((sforms (setq *sforms-trysimp-comment* (s-forms (current-goal *ps*))))
         (neg_quantform_list (setq *neg-quantform-comment*
          (gather-fnums (s-forms (current-goal *ps*)) '- nil
                        #'(lambda (sform)
                            (let ((expr (formula sform)))
                                 (has_quant expr))))))
         (pos_quantform_list (setq *pos-quantform-comment*
          (gather-fnums (s-forms (current-goal *ps*)) '+ nil
                        #'(lambda (sform)
                            (let ((expr (formula sform)))
                                 (has_quant expr)))))))
   (let ((label-hide-cmd
          '(then (label_formula_list
                   "quantified-formula"
                   ,(append neg_quantform_list pos_quantform_list))
                 (hide "quantified-formula")))
          (rewrite-unique-cmd
           (setq *rewrite-unique-comment*
           '(then (auto-rewrite-theory ,*timed-auto-unique-aux*)
                  (assert)
                  (stop-rewrite-theory ,*timed-auto-unique-aux*))))
          (rewrite-change-cmd-1
           (setq *rewrite-change-comment*
           '(then (stop-rewrite-theory ,*timed-auto-rewrite-thy-1*)
                  (auto-rewrite-theory ,*timed-auto-rewrite-thy-2*))))
          (rewrite-change-cmd-2
           (setq *rewrite-stop-comment*
           '(then (stop-rewrite-theory ,*timed-auto-rewrite-thy-2*)
                  (auto-rewrite-theory ,*timed-auto-unique-aux*)))))
   (then label-hide-cmd
```

```
        (time_etc_simp)
        (reveal "quantified-formula")
; NOTE: the following LIFT-IF is included simply so that the base case
; in lemma_3_3_3 in fischer can be handled.  If it is omitted, and
; time_vals_simp is done first, then lift-if fails to turn the CASE
; expression into an if-then-else expression.  Perhaps this is a result
; of some unexpected property of using EXPAND with :assert? NONE  ???
        (lift-if)
        (time_vals_simp)
        (repeat* (then* (lift-if) (split) (flatten) (assert)))
; NOTE: the following takes care of several cases where apply-extensionality
; was previously needed.
        rewrite-change-cmd-1
        (assert)
; NOTE: the following takes care of the cases where one previously had to
; apply a "unique_aux" lemma to the effect that if two actions of the same
; kind are equal, then their parameters are equal.
        rewrite-change-cmd-2
        (assert)
; NOTE: it is unclear why apply-extensionality seems to be needed in some
; cases; the rewrites should handle what it does !!!!  See, in particular,
; the proof of lemma_5_2 in fischer.
        (apply-extensionality)
        )))
   ""
   "Applying simple reasoning")

(defstep try_simp_again ()
   (let ((sforms (setq *sforms-trysimp-comment* (s-forms (current-goal *ps*))))
         (neg_quantform_list (setq *neg-quantform-comment*
          (gather-fnums (s-forms (current-goal *ps*)) '- nil
                        #'(lambda (sform)
                            (let ((expr (formula sform)))
                                 (has_quant expr))))))
         (pos_quantform_list (setq *pos-quantform-comment*
          (gather-fnums (s-forms (current-goal *ps*)) '+ nil
                        #'(lambda (sform)
                            (let ((expr (formula sform)))
                                 (has_quant expr)))))))
   (let ((label-hide-cmd
          `(then (label_formula_list
                    "quantified-formula"
                    ,(append neg_quantform_list pos_quantform_list))
                 (hide "quantified-formula")))
          (rewrite-init-cmd
           (setq *rewrite-init-comment*
           `(auto-rewrite-theory ,*timed-auto-rewrite-thy-1*)))
          (rewrite-unique-cmd
           (setq *rewrite-unique-comment*
           `(then (auto-rewrite-theory ,*timed-auto-unique-aux*)
                  (assert)
```

```
                (stop-rewrite-theory ,*timed-auto-unique-aux*))))
          (rewrite-change-cmd-1
           (setq *rewrite-change-comment*
           '(then (stop-rewrite-theory ,*timed-auto-rewrite-thy-1*)
                  (auto-rewrite-theory ,*timed-auto-rewrite-thy-2*))))
          (rewrite-change-cmd-2
           (setq *rewrite-stop-comment*
           '(then (stop-rewrite-theory ,*timed-auto-rewrite-thy-2*)
                  (auto-rewrite-theory ,*timed-auto-unique-aux*)))))
   (then label-hide-cmd
         rewrite-init-cmd
         (time_etc_simp)
         (try (then (repeat* (replace*)) (assert)) (skip) (skip))
         (reveal "quantified-formula")
; NOTE: the following LIFT-IF is included simply so that the base case
; in lemma_3_3_3 in fischer can be handled.  If it is omitted, and
; time_vals_simp is done first, then lift-if fails to turn the CASE
; expression into an if-then-else expression.  Perhaps this is a result
; of some unexpected property of using EXPAND with :assert? NONE  ???
         (lift-if)
; NOTE: iff solves a problem in one application (rpc_memoryimpl) where
; a = b and c = d are insufficient for ASSERT to conclude that a AND c
; = b AND d (all values being boolean).  Thus, it really is mostly a band-aid.
         (iff)
         (time_vals_simp)
         (repeat* (then (lift-if) (split) (flatten) (assert)))
; NOTE: the following takes care of several cases where apply-extensionality
; was previously needed.
         rewrite-change-cmd-1
         (assert)
; NOTE: the following takes care of the cases where one previously had to
; apply a "unique_aux" lemma to the effect that if two actions of the same
; kind are equal, then their parameters are equal.
         rewrite-change-cmd-2
         (assert)
; NOTE: it is unclear why apply-extensionality seems to be needed in some
; cases; the rewrites should handle what it does !!!!  See, in particular,
; the proof of lemma_5_2 in fischer.
         (apply-extensionality)
         (try (then (repeat* (replace*)) (assert)) (skip) (skip))
         )))
   ""
   "Applying simple reasoning")

(defstep timed_auto_forward ()
   (let ((cmd '(,*timed-auto-forward-strat*)))
   cmd)
   "" "")

(defstep try_simp_again2 ()
   (let ((sforms (setq *sforms-trysimp-comment* (s-forms (current-goal *ps*))))
```

```
        (neg_quantform_list (setq *neg-quantform-comment*
         (gather-fnums (s-forms (current-goal *ps*)) '- nil
                       #'(lambda (sform)
                           (let ((expr (formula sform)))
                                (has_quant expr))))))
        (pos_quantform_list (setq *pos-quantform-comment*
         (gather-fnums (s-forms (current-goal *ps*)) '+ nil
                       #'(lambda (sform)
                           (let ((expr (formula sform)))
                                (has_quant expr)))))))
   (let ((label-hide-cmd
          '(then (label_formula_list
                   "quantified-formula"
                   ,(append neg_quantform_list pos_quantform_list))
                 (hide "quantified-formula")))
         (forward-cmd
          (setq *simp-forward-comment* '(,*timed-auto-forward-strat*)))
         (rewrite-init-cmd
          (setq *rewrite-init-comment*
          '(auto-rewrite-theory ,*timed-auto-rewrite-thy-1*)))
         (rewrite-stop-cmd
          (setq *rewrite-stop-comment*
          '(stop-rewrite-theory ,*timed-auto-rewrite-thy-1*)))
         (rewrite-unique-cmd
          (setq *rewrite-unique-comment*
          '(then (auto-rewrite-theory ,*timed-auto-unique-aux*)
                 (assert)
                 (stop-rewrite-theory ,*timed-auto-unique-aux*))))
         (rewrite-change-cmd-1
          (setq *rewrite-change-comment*
          '(then (stop-rewrite-theory ,*timed-auto-rewrite-thy-1*)
                 (auto-rewrite-theory ,*timed-auto-rewrite-thy-2*))))
         (rewrite-change-cmd-2
          (setq *rewrite-stop-comment*
          '(then (stop-rewrite-theory ,*timed-auto-rewrite-thy-2*)
                 (auto-rewrite-theory ,*timed-auto-unique-aux*)))))
   (then label-hide-cmd
         rewrite-stop-cmd
         forward-cmd
         (try (then (repeat* (replace*)) (assert)) (skip) (skip))
         rewrite-init-cmd
         (time_etc_simp)
        ;rewrite-stop-cmd
        ;forward-cmd
        ;(try (then (repeat* (replace*)) (assert)) (skip) (skip))
         (reveal "quantified-formula")
; NOTE: the following LIFT-IF is included simply so that the base case
; in lemma_3_3_3 in fischer can be handled.  If it is omitted, and
; time_vals_simp is done first, then lift-if fails to turn the CASE
; expression into an if-then-else expression.  Perhaps this is a result
; of some unexpected property of using EXPAND with :assert? NONE  ???
```

```
        (lift-if)
; NOTE: iff solves a problem in one application (rpc_memoryimpl) where
; a = b and c = d are insufficient for ASSERT to conclude that a AND c
; = b AND d (all values being boolean).  Thus, it really is mostly a band-aid.
        (iff)
        (time_vals_simp)
       ;rewrite-init-cmd
       ;(assert)
        (repeat* (then (lift-if) (split) (flatten) (assert)))
        rewrite-stop-cmd
        forward-cmd
        (try (then (repeat* (replace*)) (assert)) (skip) (skip))
; NOTE: the following takes care of several cases where apply-extensionality
; was previously needed.
        rewrite-change-cmd-1
        (assert)
; NOTE: the following takes care of the cases where one previously had to
; apply a "unique_aux" lemma to the effect that if two actions of the same
; kind are equal, then their parameters are equal.
        rewrite-change-cmd-2
        (assert)
; NOTE: it is unclear why apply-extensionality seems to be needed in some
; cases; the rewrites should handle what it does !!!!  See, in particular,
; the proof of lemma_5_2 in fischer.
        (apply-extensionality)
       ;(try (then (repeat* (replace*)) (assert)) (skip) (skip))
        )))
   ""
   "Applying simple reasoning")

; ***                          Section 3                              ***
;
; ***        Apply-lemma strategies for timed automata.               ***

; Some of the apply-lemma strategies are specialized for application of
; state invariant lemmas.

(defstep apply_lemma (lem &rest args)
  (let ((instcmd (cons 'inst (cons -1 args)))
        (cmd (setq *cmd-comment2*
                   `(then (with-labels (lemma ,lem) ((,lem)))
                          (apply_lemma_2 ,instcmd ,lem)))))
  cmd)
  ""
  "Applying a lemma to some arguments")

(defstep apply_lemma_2 (instcmd lemma_name)
   (let ((sforms (setq *applylem2-sforms-comment*
           (s-forms (current-goal *ps*))))
         (lemma_formula_list (setq *lemma-lab-comment*
           (loop for sform in sforms
```

```
            when (string-equal (princ-to-string (car (label sform)))
                               (princ-to-string lemma_name))
            collect sform)))
          (lemma_body (setq *body-comment*
            (cond (lemma_formula_list (formula (car lemma_formula_list)))
                  (t nil))))
          (lemma_comment (setq *comment-comment*
            (cond
             ((and lemma_body *timed-auto-verbose-proofs*)
              (format nil "~a~a~a~a" "Applying the lemma
" (princ-to-string (argument lemma_body)) "
to the arguments
" (cddr instcmd)))
             (t ""))))
          (cmd (setq *apply2-cmd-comment*
            (cond (lemma_body
                    '(then (comment ,lemma_comment)
                           ,instcmd (,*timed-auto-simp-strat*)))
                  (t '(skip))))))
    cmd)
    "" "")

(defstep apply_lemma_no_comment (lem &rest args)
  (let ((instcmd (cons 'inst (cons -1 args)))
        (cmd (setq *cmd-comment2*
                   '(then (with-labels (lemma ,lem) ((,lem)))
                          ,instcmd (,*timed-auto-simp-strat*)))))
  cmd)
  ""
  "Applying a lemma to some arguments")

; poststate_strategy is used in supporting application of invariant lemmas
; to the poststate in an induction proof -- it is called indirectly by
; apply_inv_lemma

(defstep poststate_strategy (lemma_name)
  (let ((sforms (setq *sforms-comment* (s-forms (current-goal *ps*))))
        (poststate-fnums (setq *poststate-strategy-fnums-comment*
          (gather_fnums_label sforms (intern lemma_name))))
        (cmd (setq *poststate-strategy-command-comment*
             '(then (replace "poststate-definition" ,poststate-fnums rl)
                    (do_trans ,lemma_name)
                    (hide "poststate-definition")))))
  cmd)
  "" "")

(defstep apply_inv_lemma (invno &rest state+quantvars)
  (let ((cmd (setq *apply-inv-lemma-cmd-comment*
        (let (;(dummy
              ;  (setq *typecheck-comment* (typecheck
              ;       (pc-parse (car state+quantvars) 'expr))))
```

```
              (state-arg-present (is_state_var (car state+quantvars))))
        (cond (state-arg-present
               (let ((state (setq *state-comment* (car state+quantvars)))
                     (quantvars (setq *quantvars-comment*
                                      (cdr state+quantvars))))
               (setq *state-cmd-comment*
               (cond (quantvars
                      `(apply_univ_inv_lemma ,invno ,quantvars ,state))
                     (t `(apply_simple_inv_lemma ,invno ,state))))))
              (t (setq *no-state-cmd-comment*
                 (cond (state+quantvars `(apply_univ_inv_lemma ,invno
                                                  ,state+quantvars))
                       (t `(apply_simple_inv_lemma ,invno)))))))))
  cmd)
  ""
  "Applying a state invariant")

(defstep apply_simple_inv_lemma (invno &optional statevar)
  (let ((lemma_name (setq *simple-lemmaname-comment*
                   (format nil "~a~a" "lemma_" invno)))
        (theorem_name (setq *simple-theoremname-comment*
                   (format nil "~a~a" "theorem_" invno)))
        (inv_name (format nil "~a~a" "Inv_" invno))
        (state (cond (statevar) (t "prestate"))))
  (let ((poststate-cmd
        (setq *simple-poststate-cmd-comment*
        (cond ((string-equal state "poststate")
               `(then (reveal "poststate-definition")
                      (poststate_strategy ,lemma_name)))
              (t '(skip))))))
  (then (try (apply_lemma_no_comment lemma_name state)
             (skip)
             (apply_lemma_no_comment theorem_name state))
        (apply (then (split -1 :depth 1) (assert)))
        (expand inv_name)
        (apply_simple_inv_lemma_2 lemma_name)
        poststate-cmd)))
  ""
  "Applying the appropriate invariant lemma")

(defstep apply_simple_inv_lemma_2 (lemma_name)
  (let ((sforms (setq *simple-sforms-comment* (s-forms (current-goal *ps*))))
        (lemma_body (setq *simple-body-comment*
         (formula (car (loop for sform in sforms
                   when (string-equal (princ-to-string (car (label sform)))
                                      (princ-to-string lemma_name))
                   collect sform)))))
         (lemma_comment (setq *simple-comment-comment*
          (cond
           (*timed-auto-verbose-proofs*
            (format nil "~a~a" "Applying the lemma
```

```
" (princ-to-string (argument lemma_body))))
          (t ""))))
       (simp_cmd (list *timed-auto-simp-strat*))
       (cmd (setq *apply2-cmd-comment* '(then (comment ,lemma_comment)
                  (,*timed-auto-simp-strat*)))))
       cmd)
  "" "")

(defstep apply_univ_inv_lemma (invno quantvars &optional statevar)
  (let ((lemma_name (setq *univ-lemmaname-comment*
        (format nil "~a~a" "lemma_" invno)))
       (theorem_name (setq *univ-theoremname-comment*
        (format nil "~a~a" "lemma_" invno)))
       (inv_name (format nil "~a~a" "Inv_" invno))
       (state (setq *univ-state-comment* (cond (statevar) (t "prestate"))))
       (inst_cmd (cons 'inst (cons '-1 quantvars)))
       (dummy (setq *inst-comment* (princ-to-string inst_cmd))))
  (let ((poststate-cmd
        (setq *univ-poststate-cmd-comment*
              (cond ((string-equal state "poststate")
                     '(then (reveal "poststate-definition")
                            (poststate_strategy ,lemma_name)))
                    (t '(skip))))))
  (then (try (apply_lemma_no_comment lemma_name state)
             (skip)
             (apply_lemma_no_comment theorem_name state))
       (apply (then (split -1 :depth 1)(assert)))
       (expand inv_name)
       (apply_univ_inv_lemma_2 lemma_name inst_cmd)
       poststate-cmd)))
  ""
  "Applying and instantiating the appropriate invariant lemma")

(defstep apply_univ_inv_lemma_2 (lemma_name inst_cmd)
  (let ((sforms (setq *univ-sforms-comment* (s-forms (current-goal *ps*))))
       (lemma_body (setq *univ-body-comment*
        (formula (car (loop for sform in sforms
                   when (string-equal (princ-to-string (car (label sform)))
                                      (princ-to-string lemma_name))
                   collect sform)))))
       (lemma_comment (setq *univ-comment-comment*
        (cond
         (*timed-auto-verbose-proofs*
          (format nil "~a~a" "Applying the lemma
" (princ-to-string (argument lemma_body))))
          (t ""))))
       (simp_cmd (list *timed-auto-simp-strat*))
       (cmd (setq *apply2-cmd-comment* '(then (comment ,lemma_comment)
                  ,inst_cmd
                  (,*timed-auto-simp-strat*)))))
       cmd)
```

```
  "" "")

; ***                          Section 4                                ***
;
; ***                  Other standard TAME steps.                       ***

(defstep apply_ind_hyp (&rest var)
   (let ((simplification_strat *timed-auto-simp-strat*)
         (instcmd (setq *apply-ind-hyp-inst-comment*
            (cons 'inst (cons "inductive-hypothesis" var))))
         (cmd `(,simplification_strat)))
   (then (reveal "inductive-hypothesis")
         instcmd
         cmd))
   "" "")

; The strategy apply_specific_precond is broken into two parts in order
; to permit the specific precondition to be expanded and the resulting
; formula to be printed as a comment.

(defstep apply_specific_precond ()
   (let ((sforms
          (setq *sforms-precond-comment* (s-forms (current-goal *ps*)))))
   (let ((simp-flag (setq *simp-flag-comment* (has_specific_precond sforms))))
   (then (with-labels (try (expand "enabled") (flatten) (skip))
                      (("general-precondition" "specific-precondition"
                        "OKstate?-precondition")) :push? T)
         (expand "enabled_specific")
         (apply_specific_precond_2 simp-flag))))
  "" "")

(defstep apply_specific_precond_2 (simp-flag)
   (let ((sforms
          (setq *sforms-precond-comment2* (s-forms (current-goal *ps*))))
         (simplification_strat *timed-auto-simp-strat*))
   (let ((precond_sform (get_form_label sforms '|specific-precondition| )))
   (let ((precond_form (setq *precond-form-comment*
          (cond (precond_sform (argument (formula precond_sform)))
                (t nil)))))
   (let ((cmd (setq *precond-cmd-comment*
          (cond (precond_form `(then
                 (comment
                   ,(cond (*timed-auto-verbose-proofs*
                           (format nil "~a~a" "Applying the precondition
" (princ-to-string precond_form)))
                          (t "")))
                 (flatten_labelled_formula '|specific-precondition| )
                 (,simplification_strat)))
                (simp-flag `(,simplification_strat))
                (t '(skip))))))
    cmd))))
```

```
  "" "")

(defstep apply_general_precond () (expand "enabled_general") "" "")

(defstep use_OKstate ()
   (let ((simp_cmd (list *timed-auto-simp-strat*)))
   (then (expand "OKstate?")
         simp_cmd
         (do_trans "OKstate?-precondition")))
  ""
  "using the OKstate? precondition")

(defstep suppose (x)
   (let ((simp_cmd (list *timed-auto-simp-strat*))
         (suppstring (setq *supp-comment*
          (format nil "~a~a" "Suppose " x)))
         (nsuppstring (setq *supp-not-comment*
          (format nil "~a~a~a" "Suppose not [" x "]"))))
   (branch (with-labels (case x) (("Suppose") ("Suppose not")))
           ((then simp_cmd (comment suppstring))
            (then simp_cmd (comment nsuppstring)))))
   "" "")

;; The strategy const_facts introduces the facts about the constants from
;; the axiom "const_facts" in the template.  It has two segments, to allow
;  the body of the axiom "const_facts" to be expanded and printed as a comment.

(defstep const_facts ()
  (then (with-labels (lemma "const_facts") (("const_facts")))
        (const_facts_2))
  ""
  "Adducing facts about the constants")

(defstep const_facts_2 ()
   (let ((sforms
          (setq *sforms-const-comment* (s-forms (current-goal *ps*))))
         (simplification_strat *timed-auto-simp-strat*)
         (const_form (setq *const-form-comment*
          (argument (formula
           (get_form_label sforms '|const_facts|)))))
         (cmd (setq *const-cmd-comment* '(then
                (comment ,(format nil "~a~a" "Applying the facts about the constants:
" (princ-to-string const_form)))
                (flatten_labelled_formula '|const_facts|)
                (,simplification_strat)))))
    cmd)
  "" "")

(defstep do_trans (&optional formnum)
   (let ((cmd (cond (formnum
                    '(then (expand "trans" ,formnum)
```

```
                              (,*timed-auto-simp-strat*)
                              (lift-if ,formnum)(assert)(assert)))
                        (t '(then (expand "trans")
                                  (,*timed-auto-simp-strat*)
                                  (lift-if)(assert)(assert))))))
     cmd)
    ""
    "Computing the transition")

(defstep compute_poststate (&optional lab)
   (let ((sforms (setq *sforms-comment* (s-forms (current-goal *ps*))))
         (formnums (setq *compute-post-formnums-comment*
            (gather_fnums_label sforms (intern lab))))
         (cmd (cond (formnums
                     '(then (reveal "poststate-definition")
                            (replace "poststate-definition" ,formnums rl)
                            (hide "poststate-definition")
                            (do_trans ,lab)))
                    (t '(then (reveal "poststate-definition")
                              (replace "poststate-definition" * rl)
                              (hide "poststate-definition")
                              (do_trans))))))
     cmd)
    ""
    "Computing the poststate")

; The strategy focus_on differs from the "scr" version in that the value
; of the poststate is not repeatedly substituted.  Note that focus_on does
; not work well unless a label is passed as argument.

;(defstep focus_on (lab)
;   (let ((cmd '(apply (repeat*
;                      (then (split ,lab) (lift-if ,lab) (flatten)(assert))))))
;   cmd)
;   "" "")

(defstep focus_on (lab)
   (let ((cmd '(apply (repeat*
                      (then (split ,lab) (lift-if ,lab)
                            (flatten_labelled_formula ,lab) (assert))))))
   cmd)
   "" "")

(defstep inst_in (lab &rest args)
   (let ((cmd (setq *inst_in-cmd-comment* '(then (focus_on ,lab)
                     ,(cons 'inst (cons lab args))
                     (,*timed-auto-simp-strat*)
                     ;(focus_on ,lab)
                     ))))
   cmd)
   "" "")
```

```
(defstep skolem_in (lab &rest args)
   (let ((cmd (setq *skolem_in-cmd-comment* '(then (focus_on ,lab)
                    (skolem ,lab ,args)
                    (,*timed-auto-simp-strat*)
                   ;(focus_on ,lab)
                    ))))
   cmd)
   "" "")

(defstep verbose ()
   (let ((dummy (setq *timed-auto-verbose-proofs* t)))
   (skip))
   "" "")

(defstep nonverbose ()
   (let ((dummy (setq *timed-auto-verbose-proofs* nil)))
   (skip))
   "" "")

; ***                          Section 5                                  ***
;
; *** Definitions and strategies to support reasoning about epsilon ***

(defun attach_arg (x y)
   (if (null y)
       (format nil "~a~a" x ")")
       (if (null (cdr y))
           (format nil "~a~a~a~a" x ", " (car y) ")")
           (format nil "~a~a~a" x ", " (attach_arg (car y) (cdr y))))))

(defun make_pref_expr (eps_pred_name &rest eps_pred_args)
   (if (null eps_pred_args)
       eps_pred_name
       (format nil "~a~a~a" eps_pred_name "("
                   (attach_arg (car eps_pred_args) (cdr eps_pred_args)))))

(defun grab_epsilon_expr (sforms predname)
  (cond (sforms (cond ((grab_epsilon_expr_2 (car sforms) predname))
                      (t (grab_epsilon_expr (cdr sforms) predname))))
        (t nil)))

(defun grab_epsilon_expr_2 (sform predname)
  (let ((epsilon_expr nil)
        (expr (formula sform)))
  (mapobject #'(lambda (x) (if epsilon_expr t
                               (when (and (typep x 'application)
                                          (tc-eq (id (operator x)) '|epsilon|)
                                          (tc-eq
                                            (cond ((application? (argument x))
                                                   (id (operator (argument x))))
                                                  ((name-expr? (argument x))
```

```
                                                    (id (argument x))))
                                              (intern predname)))
                                        (setq epsilon_expr x)
                                        t)))
            expr)
   epsilon_expr))

(defstep use_epsilon (eps_pred_name &rest eps_pred_args)
   (let ((sforms (setq *sforms-comment* (s-forms (current-goal *ps*))))
         (eps_expr (setq *epsilon-comment*
           (grab_epsilon_expr sforms eps_pred_name)))
         (inst_pred (setq *pred-comment*
           (cond ((setq *args-comment* eps_pred_args)
                  (setq *args-comment2*
                  (eval (cons 'make_pref_expr
                              (cons eps_pred_name eps_pred_args)))))
                 (t (princ-to-string (argument eps_expr))))))
         (eps_constraints_comment
           (format nil "~a~a~a" "Introducing the constraints on
epsilon(" inst_pred ")"))
         (eps_obligation_comment
           (format nil "~a~a" "Proof that there is a value satisfying
" inst_pred))
         (eps_type (setq *type-comment* (type eps_expr)))
         (eps_lemma (setq *epslemma-comment*
          (format nil "~a~a~a" "epsilon_ax[" (princ-to-string eps_type) "]")))
         (eps_lemma_cmd (setq *epslemma-comment2* `(lemma ,eps_lemma)))
         (expand_cmd (setq *expand-comment* `(expand ,eps_pred_name 1))))
   (then eps_lemma_cmd
         (inst -1 inst_pred)
         (branch (with-labels
                  (split -1)
                  (("epsilon axiom")
                   ("epsilon axiom existence proof obligation")))
                 ((then (expand eps_pred_name -1 1) (flatten)
                        (comment eps_constraints_comment))
                  (then expand_cmd
                        (comment eps_obligation_comment))))))
  ""
  "Adducing facts about an epsilon expression")

(defstep epsilon_witness (witness)
   (let ((simplification_strat *timed-auto-simp-strat*)
        (simp-cmd `(,simplification_strat)))
   (then (inst "epsilon axiom existence proof obligation" witness)
         simp-cmd))
   "" "")

; ***                          Section 6                                ***
;
; ***     Strategies to compensate for missing PVS features.            ***
```

```
(defstep modus_ponens_probe (fnum)
  (let ((sform (setq *sform-comment*
         (cond ((< fnum 0)
                (nth (- (abs fnum) 1)
                     (gather_negsforms (s-forms (current-goal *ps*)))))
               (t
                (nth (- fnum 1)
                     (gather_possforms (s-forms (current-goal *ps*))))))))
        (fnum-raw-labels (label sform))
        (labels (reverse (mapcar #'princ-to-string fnum-raw-labels)))
        (lastlabel (setq *lastlabel-comment*
         (cond (labels (car labels)) (t nil))))
        (restlabels (setq *restlabels-comment*
         (cond (labels (cdr labels)) (t nil))))
        (labelcmd (setq *label-cmd-comment*
         (cons 'then
               (loop for lab in labels collect `(label ,lab "temp" :push? T)))))
        (form (setq *form-comment* (formula sform)))
        (posform (setq *posform-comment*
         (if (negation? form) (args1 form) form)))
        (concl (setq *conclusion-comment*
         (if (implication? posform) (args2 posform)
             (if (conjunction? posform) (args2 posform) posform))))
        (conclstring (princ-to-string concl))
        (next-fnum (setq *next-fnum-comment*
         (cond ((< fnum 0) (- (+ (abs fnum) 1))) (t (+ fnum 1)))))
        (casecmd (setq *case-cmd-comment*
         `(with-labels (case ,conclstring) (("temp")))))
        (cmd (cond ((< fnum 0)
                    `(try (branch ,casecmd
                            (,labelcmd
                             (then (flatten) (assert) (lift-if) (prop)
                                   (assert) (fail))))
                          (hide ,next-fnum)
                          (skip)))
                   (t
                    `(try (branch ,casecmd
                            ((then (flatten) (assert) (lift-if) (prop)
                                   (assert) (fail))
                             ,labelcmd))
                          (hide ,next-fnum)
                          (skip))))))
    cmd)
    ""
    "Attempting to eliminate an hypothesis")

(defstep cancel_formulas_once ()
   (let ((sforms (setq *sforms-comment* (s-forms (current-goal *ps*))))
         (implication_list (setq *implication-comment*
          (gather-fnums (s-forms (current-goal *ps*)) '* nil
                        #'(lambda (sform)
```

```
                              (let ((expr (formula sform)))
                                (or (and (negation? expr)
                                         (implication? (args1 expr)))
                                    (and (not (negation? expr))
                                         (conjunction? expr))))))))
          (dummy1 (setq *mapcar-comment*
           (mapcar #'(lambda (expr) (or (and (negation? expr)
                                             (implication? (args1 expr)))
                                        (and (not (negation? expr))
                                             (conjunction? expr))))
           (mapcar #'formula (s-forms (current-goal *ps*))))))
          (thenlist (loop for formnum in implication_list collect
                          `(modus_ponens_probe ,formnum)))
          (cmd `(apply ,(cons 'then thenlist)))
          (dummy (setq *compute-comment3* (princ-to-string cmd))))
         cmd)
   "" "")

(defstep cancel_formulas ()
   (repeat* (cancel_formulas_once))
   "" "")

(defstep specialize_induction_to (&rest vars)
   (let ((cmd `(then (skolem "inductive-conclusion" ,vars)
                     ,(cons 'inst (cons "inductive-hypothesis" vars)))))
        cmd)
   "" "")

(defstep specialize_induction_to_2 (&rest vars)
   (let ((cmd `(then (skolem_in "inductive-conclusion" ,vars)
                     ,(cons 'inst_in (cons "inductive-hypothesis" vars))
                     (prop_probe))))
        cmd)
   "" "")

; ***                          Section 7                                ***
;
; ***              Miscellaneous auxiliary strategies.                  ***

; flatten_labelled_formula completely flattens the formula with label lab,
; labelling the resulting parts in the order they appear in the original
; formula.  All parts also retain their original label.

; A peculiarity of this strategy is that it uses a longer label list than
; should be necessary.  For some reason, the :push? T does not work on the
; last label in the list.  Therefore, a new label |dummy| is appended to the
; list of labels.

(defstep flatten_labelled_formula (lab)
   (let ((sforms
          (setq *sforms-flatten-comment* (s-forms (current-goal *ps*)))))
```

```
   (let ((labval (intern (eval lab))))
   (let ((fnums
          (setq *fnums-flatten-comment*
                (gather_fnums_label sforms labval))))
   (let ((cmd (setq *flatten-cmd-comment*
          (cons 'then
           (let ((labcount 0) (lablist nil))
           (loop for x in fnums do
                ;(setq labcount (+ labcount 1))
                 (setq lablist (setq *fl-lablist-comment*
                   (loop for y from 1 to
                      (setq *fl-len-comment* (flatten_length x sforms)) collect
                         (format nil "~a~a~a"
                                 labval "_part_"
                                 (setq labcount (+ labcount 1))))))
                 collect `(with-labels (flatten ,x)
                           (,(append lablist '(|dummy|)))
                           :push? T)))))))
   cmd))))
   "" "")

; label_formula_list expects a list fnums of formula numbers, and a label
; lab.  It pushes lab as an extra label on each indicated formula in the
; sequent.

(defstep label_formula_list (lab fnums)
   (let ((labval (eval lab)))
   (let ((cmd (setq *label-list-comment*
          (cons 'then
           (loop for x in fnums
                 collect `(label ,labval ,x :push? T))))))
   cmd))
   "" "")

; The strategy match_condition is used to simplify reasoning about an
; IF-THEN-ELSE assertion.  It can sometimes circumvent splitting; when
; it does not, it can make the result of splitting more "natural".

(defstep match_condition (fnum)
  (then (split fnum) (flatten) (assert))
  ""
  "Attempting to eliminate a condition")

; The strategy modus_ponens is used to avoid splitting an assertion having
; a complex hypothesis identical to another assertion present.

(defstep modus_ponens (fnum)
  (branch (split fnum) ((skip) (assert)))
  ""
  "Attempting to eliminate an hypothesis")
```

```
;; The strategy my_assert is used to capture the current s-forms at almost
;; any stage of a proof -- since "assert" almost always succeeds.  It exists
;; for strategy experimentation purposes only.

(defstep my_assert ()
  (let ((sforms (setq *sforms-comment* (s-forms (current-goal *ps*)))))
       (assert))
  "" "")

; ***                          Section 8                                ***
;
; ***       Strategies for the timed_auto version of opspec.            ***

(defstep auto_proof_opspec_timed_auto (inv)
  (then (branch (time (auto_cases inv))
                ((then (base_case_timed_auto inv)(opspec_simp_probe)(postpone))
                 (branch (induct_cases inv)
                         ((then (reduce_case_timed_auto_vars_exp inv "t_1")
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_timed_auto_vars_exp inv "r_1")
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_timed_auto_vars_exp inv "r_1")
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_timed_auto_vars_exp inv "r_1")
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_timed_auto_no_var_exp inv)
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_timed_auto_no_var_exp inv)
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_timed_auto_no_var_exp inv)
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_timed_auto_no_var_exp inv)
                                (opspec_simp_probe)(postpone)))))))
  ""
  "Taking care of the standard steps in the proof")

(defstep base_case_timed_auto (inv)
  (then* (delete 2)
         (expand "base")
         (skolem 1 "prestate")
         (flatten)
         (expand "start")
         (flatten)
         (expand "basic_start")
         (expand inv))
  ""
  "Simplifying the auto base case")

(defstep reduce_case_timed_auto_vars_exp (inv vars)
  (then* (delete 2)
         (skolem 1 (vars))
```

```
        (skolem 1 ("prestate"))
        (flatten)
        (expand "enabled")
        (expand "trans")
        (expand "basic_trans")
        (expand inv))
  ""
  "Applying the standard simplification")

(defstep reduce_case_timed_auto_no_var_rew (inv)
  (then* (delete 2)
        (skolem 1 ("prestate"))
        (flatten)
        (rewrite "enabled")
        (rewrite "trans")
        (expand "basic_trans")
        (expand inv))
  ""
  "Applying the standard simplification")

(defstep reduce_case_timed_auto_no_var_exp (inv)
  (then* (delete 2)
        (skolem 1 ("prestate"))
        (flatten)
        (expand "enabled")
        (expand "trans")
        (expand "basic_trans")
        (expand inv))
  ""
  "Applying the standard simplification")

(defstep normalize_atexecs_timed_auto ()
   (then (auto-rewrite-theory
        "timed_auto_thy [ basic_states, actions, nu, nu?, timeof,
                          basic_start, first_start, last_start,
                          basic_trans, first_trans, last_trans,
                          enabled_specific, OKstate?]")
        (apply (do-rewrite)))
   "" "")

(defstep do_trans_opspec_timed_auto ()
   (then (expand "trans")
        (expand "basic_trans")(expand "first_trans")(expand "last_trans")
        (| opspec_simp| )(lift-if)(assert)(assert))
   "" "")

; ***                          Section 9                                ***
;
; ***    General strategies useful in reasoning about atexecs.          ***

; The strategy put_glb finds the time index of the last indexed time in
```

```
; an atexec that is less than or equal to the particular non-negative-real
; valued bound "timebound", and gives it an associated name.

(defstep put_glb (atexec timebound)
  (let ((x (format nil "~a~a" timebound "_glb"))
        (y timebound)
        (z atexec))
       (put_glb_2 x y z))
  "" "")

(defstep put_glb_2 (boundname timebound atexec)
  (let ((x (list atexec timebound))
        (y (list boundname)))
  (then (apply_lemma "glb_fact" x) (skolem -1 y) (flatten)))
  "" "" )

; The strategy get_reachables adduces the fact of reachability for
; states in an atexec near time index "index", under various aliases.

(defstep get_reachables (atexec index)
   (let ((x (list atexec index)))
        (then (apply_lemma "reachable_states" x) (flatten)))
   "" "")

; The strategy trans_facts adduces the relatedness of states, under various
; aliases, via a transition in an atexec near time index "index".

(defstep trans_facts (atexec index)
   (let ((x (list atexec index)))
        (then (apply_lemma "trans_facts" x)
        (flatten) (assert) (flatten)))
   "" "")

; The strategy normalize_atexecs converts all time points and state points
; of an admissible timed execution to a normal form, so that equalities
; may be inferred.

(defstep normalize_atexecs ()
   (then (auto-rewrite-theory
          "atexecs_strat_aux[states,actions,start,now,step?,nu]")
         (do-rewrite))
   "" "")

; The strategy time_order is used to infer an inequality between time
; indices from the same inequality between the indexed times.

(defstep time_order (atexec n1 n2)
   (let ((x (list atexec n1 n2)))
        (then (apply_lemma "time_relation" x) (flatten) (simplify)))
   "" "")
```

```
; same_states_tcc is a special tcc  strategy useful in proving the tccs for
; the lemmas about admissible timed traces used to support normalized_atexecs:

(defstep same_states_tcc (atexec leftend rightend)
  (let ((timeseq (format nil "~a~a~a" "t(" atexec ")"))
        (trajseq (format nil "~a~a~a" "w(" atexec ")")))
  (then (skosimp)
        (expand "interval")
        (apply (then (typepred atexec) (hide -1 -3 -4) (inst-cp -1 leftend)
                     (inst -1 rightend)))
        (apply (then (typepred timeseq) (hide -1) (inst -1 leftend rightend)))
        (apply (then (typepred trajseq) (inst -1 leftend)))
        (expand "ltime")
        (assert)))
  "" ""}

; ***       New strategies inspired by the Utility Prop. Prf.       ***

(defstep name_last_event (ename pname atexec state)
  (let ((exists_case_body (format nil "~a~a~a~a~a~a~a~a~a~a~a"
                            "(exists (n: posnat): (precedes_state("
                            atexec ")(n ," state ") & " pname "(" atexec ", "
                            state ", n)))"))
        (pname_before_s_index (format nil "~a~a~a~a"
                                pname "_before_" state "_index"))
        (ename_index (format nil "~a~a" ename "_index"))
        (ename_event (format nil "~a~a~a~a~a~a"
                       "pi(" atexec ")(" ename "_index" ")" )))
  (then (branch (case exists_case_body)
                ((then (skolem -1 pname_before_s_index)
                       (flatten -1)
                       ;(apply_lemma "last_event" (atexec state pname))
                       (branch (apply_lemma "last_event" (atexec state pname))
                               ((then (simplify)
                                      (inst -1 pname_before_s_index)
                                      (modus_ponens -1)
                                      (hide -2 -3)
                                      (skolem -1 ename_index)
                                      (flatten -1)
                                      (hide -1)
                                      (name ename ename_event))
                                (skip))))
                 (skip)))))
  ""
  "Let ~a be the last event with Property ~a in ~a that occurs before ~a.")

(defstep start_state_props_opspec (atexec)
  (then (turnoff_rewrites "opspec")
        (typepred atexec)
        (turnon_rewrites "opspec")
        (hide -2 -3 -4)
```

```
        (expand "start")
        (normalize "opspec")
        (replace -1)
        (| opspec_simp| )
        (assert))
  ""
  "Recognizing the start state of ~a and invoking its opspec properties.")

(defstep trajectory_order (atexec state)
  (let ((traj_index (format nil "~a~a" state "_traj_index"))
        (precedes_facts_1_args (list atexec traj_index state)))
  (then (expand "in_atexec" -1)
        (skolem -1 traj_index)
        (apply_lemma "precedes_facts_1" precedes_facts_1_args)
        (flatten)
        (assert)))
  ""
  "Adducing the fact that, in ~a, ~a occurs between the
   endpoints of its trajectory.")

;(defstep atexec_order (atexec index1 index2)

(defstep check_enabled_specific_opspec (&optional formnum)
   (let ((cmd (cond (formnum
                    `(then (expand "enabled_specific" ,formnum)
                           (opspec_simp ,formnum) (lift-if ,formnum)
                           (assert)(assert)))
                    (t '(then (expand "enabled_specific")(| opspec_simp| )
                              (lift-if)(assert)(assert))))))
    cmd)
   ""
   "Expanding and simplifying the definition of enabled_specific.")

(defstep get_enabled_specific (atexec index)
   (let ((fromindex (format nil "~a~a" index " - 1")))
   (then (turnoff_rewrites "opspec")
         (typepred atexec)
         (turnon_rewrites "opspec")
         (hide -1 -2 -4)
         (expand "step?")
         (inst -1 fromindex)
         (simplify)
         (flatten)
         (hide -2)
         (expand "enabled" -1)
         (flatten)
         (hide -1 -3)))
   ""
   "Retrieving the information that in ~a, the ~a-th action is specifically
    enabled.")
```

```
;(defstep normalize ()
;   (then (auto-rewrite-theory
;          "atexecs_strat_aux [states,actions,start,now,step?,nu]")
;         (apply (do-rewrite)))
;   "" "")

; A present from Shankar .....

(defun listify (x) (if (listp x) x (list x)))

(defstep auto-rewrite-theory-with-importings
  (name &optional exclude-theories importchain? exclude defs
      always? tccs?)
  (let ((name (pc-parse name 'modname))
      (theory-name (resolve-theory-name name))
      (exclude-theories (listify exclude-theories))
      (exclude-theory-names
       (mapcar #'(lambda (x) (pc-parse x 'modname))
         exclude-theories))
      (theory (get-theory name))
      (usings (if importchain?
                (mapcar #'(lambda (x)(cadr x))
                  (all-usings theory))
                (immediate-usings theory)))
      (included-usings
       (loop for z in (cons theory-name usings)
             unless (member z exclude-theory-names
                            :test
                            #'(lambda (u v)
                                (if (actuals v)
                                  (ps-eq u v)
                                  (same-id u v))))
             collect z))
      (theories
       (loop for x in included-usings
             collect
             (list x :exclude exclude :defs defs
                     :always? always? :tccs? tccs?))))
    (auto-rewrite-theories$ :theories theories))
  "Installs rewrites in theory NAME and along with any theories
imported by NAME.  The full import chain of theories can be
installed by supplying the IMPORTCHAIN? flag as T.  Theories named
in EXCLUDE-THEORIES are ignored.  The other arguments are similar
to those of auto-rewrite-theory and apply uniformly to each of
the theories to be installed."
  "Rewriting with ~a and imported theories therein")

(defstep stop-rewrite-theory-with-importings
  (name &optional exclude-theories importchain?)
  (let ((name (pc-parse name 'modname))
      (theory-name (resolve-theory-name name))
```

```
      (exclude-theories (listify exclude-theories))
      (exclude-theory-names
       (mapcar #'(lambda (x) (pc-parse x 'modname))
         exclude-theories))
      (theory (get-theory name))
      (usings (if importchain?
                  (mapcar #'(lambda (x)(cadr x))
                    (all-usings theory))
                  (immediate-usings theory)))
      (included-usings
       (loop for z in (cons theory-name usings)
             unless (member z exclude-theory-names
                            :test
                            #'(lambda (u v)
                                (if (actuals v)
                                  (ps-eq u v)
                                  (same-id u v))))
             collect z))
      (theories included-usings))
   (stop-rewrite-theory$ :theories theories))
  "Un-Installs rewrites in theory NAME and along with any theories
imported by NAME.  The full import chain of theories can be
un-installed by supplying the IMPORTCHAIN? flag as T.  Theories named
in EXCLUDE-THEORIES are ignored.  The other arguments are similar
to those of auto-rewrite-theory and apply uniformly to each of
the theories to be installed."
  "Stopping Rewriting with ~a and imported theories therein")

; End present from Shankar .....

(defstep turnon_rewrites (thy_name)
   (let  ((strat_thy (format nil "~a~a" thy_name "_strat_aux"))
          (non_strat_thy (format nil "~a~a" thy_name "_atexecs_aux")))
   (then (auto-rewrite-theory-with-importings
          strat_thy :exclude-theories non_strat_thy)))
   ""
   "Turning on the atexecs rewrites associated with ~a.")

(defstep turnoff_rewrites (thy_name)
   (let  ((strat_thy (format nil "~a~a" thy_name "_strat_aux"))
          (non_strat_thy (format nil "~a~a" thy_name "_atexecs_aux")))
   (then (stop-rewrite-theory-with-importings
          strat_thy :exclude-theories non_strat_thy)))
   ""
   "Turning off the atexecs rewrites associated with ~a.")

(defstep normalize (thy_name)
   (let  ((strat_thy (format nil "~a~a" thy_name "_strat_aux"))
          (non_strat_thy (format nil "~a~a" thy_name "_atexecs_aux")))
   (then (auto-rewrite-theory-with-importings
          strat_thy :exclude-theories non_strat_thy)
```

```
        (apply (do-rewrite))))
   ""
   "Normalizing the representation of times and states for ~a.")

(defstep expand_state_preds (&optional formnum)
   (let ((cmd (cond (formnum '(then (expand "predIMPLIES" ,formnum)
                                    (expand "predAND" ,formnum)
                                    (expand "predNOT" ,formnum)
                                    (expand "fstate_precedes" ,formnum)
                                    (expand "precedes_state" ,formnum)
                                    (expand "gate_status_up" ,formnum)))
                    (t '(then (expand "predIMPLIES")
                              (expand "predAND")
                              (expand "predNOT")
                              (expand "fstate_precedes")
                              (expand "precedes_state")
                              (expand "gate_status_up"))))))
    cmd)
   ""
   "Expanding the definitions of the standard state predicates and
state predicate combinators.")


; ***                          Section 10                              ***
;
; ***        Grabbing the application-specific  strategies.            ***

(load "local-strategies")
```

Appendix 4 : Example Template Instantiation, Local Theories, and Local Strategies.

The first section of this Appendix contains, as an example template instantiation, the definition of the Steam Boiler Controller studied in [AH_97a]. This definition has gone through several phases. The first phase was an encoding of the specification in [LL_96a]. The second phase incorporated modifications that were conjectured as reasonable corrections once (an older version of) TAME had uncovered some typos and inconsistencies in the original specification. The third and final phase incorporates the corrections provided by the authors of [LL_96a] in [LL_96b]. Relics of the first two phases are commented out using %%.

The remaining two sections contain the local theories and strategies for the steam boiler controller application needed to support the strategies in the current version of TAME. Although many of the lemmas in the theories in the second section have "rewrite" in their name and are currently being used as rewrites, it is planned that ultimately their main use will be in forward chaining. Since PVS has an AUTO-REWRITE-THEORY command but not an AUTO-FORWARD-CHAIN-THEORY command, one or more additions to the local strategies file will be required to accomplish this forward chaining. A future version of TAME will generate the local theories and strategies from the timed automaton template instantiation; at present, this must be tediously done by hand.

Appendix 4.1 : Instantiating the Template for a Boiler Control System.

```
boilersys_decls: THEORY
  BEGIN
    timed_auto_lib: LIBRARY = "../timed_auto_lib";
    IMPORTING timed_auto_lib@time_thy
    IMPORTING timed_auto_lib@real_thy
    I: posreal;
    S: posreal;
    U_1, U_2: posreal;
    M_1, M_2: posreal;
    W: posreal;
    P: posreal;
    num_pumps: posnat;
    C: posreal;
    delta_LOW(sr_old:nonnegreal,sr_new:nonnegreal,t:(fintime?)):nonnegreal;
    delta_HIGH(sr_old:nonnegreal,sr_new:nonnegreal,t:(fintime?)):nonnegreal;
    num_pumps: TYPE = {n:nat | n <= num_pumps};
    steam_rate: TYPE = {r:nonnegreal | r <= W};
  %% water_level: TYPE = {r:nonnegreal | r <= C};
    water_level: TYPE = real;
    max_pumps_after_set:num_pumps = num_pumps;
    min_pumps_after_set:num_pumps = 0;
    max(x1,x2:real):real = IF x1 > x2 THEN x1 ELSE x2 ENDIF;
  %% min_steam_water(sr:steam_rate):nonnegreal = max(0,(sr - U_2*I/2)*I);
    min_steam_water(sr:steam_rate):nonnegreal =
      IF sr < U_2*I THEN sr*sr/(2*U_2) ELSE sr*I - U_2*I*I/2 ENDIF;
  %% min_steam_water_est was not in the original specification.
    min_steam_water_est(sr:steam_rate):nonnegreal =
```

```
    IF sr < U_1*I THEN sr*sr/(2*U_1) ELSE sr*I - U_1*I*I/2 ENDIF;
  max_steam_water(sr:steam_rate):nonnegreal = (sr + U_1*I/2)*I;

  const_facts: AXIOM (M_1 < M_2 & M_2 <= C & S < I);
%%                  & M_1 + U_1*I*I/2 < M_2 - P*(I-S)*num_pumps
%%                  & U_1*I < W);

%% Lemmas lemma_1_1 through lemma_1 are facts that can be proved about
%% delta_LOW and delta_HIGH from appropriate definitions of these functions,
%% using various facts from the Calculus.  Here, they are simply made axioms.

  lemma_1_1: AXIOM (FORALL (a,b:nonnegreal, u:(fintime?)):
    delta_LOW(a,b,u) <= delta_HIGH(a,b,u));

  lemma_1_2: AXIOM (FORALL (a,b:nonnegreal, u:(fintime?)):
%%  delta_LOW(a,b,u) >= max(a*dur(u) - U_2*dur(u)*dur(u)/2, a*a/(2*U_2)) &
%%  max(a*dur(u) - U_2*dur(u)*dur(u)/2, a*a/(2*U_2)) >= 0);
    delta_LOW(a,b,u) >= IF a < U_2*dur(u) THEN a*a/(2*U_2)
                        ELSE a*dur(u) - U_2*dur(u)*dur(u)/2 ENDIF);

  lemma_1_3: AXIOM (FORALL (a,b:nonnegreal, u:(fintime?)):
%%  delta_LOW(a,b,u) >= max(b*dur(u) - U_1*dur(u)*dur(u)/2, 0));
    delta_LOW(a,b,u) >= IF b < U_1*dur(u) THEN b*b/(2*U_1)
                        ELSE b*dur(u) - U_1*dur(u)*dur(u)/2 ENDIF);

  lemma_1_4: AXIOM (FORALL (a,b,c:nonnegreal, t,u:(fintime?)):
    delta_LOW(a,b,u) + delta_LOW(b,c,t) >= delta_LOW(a,c,t+u));

  lemma_1_5: AXIOM (FORALL (a,b:nonnegreal, u:(fintime?)):
    (a + b)*dur(u)/2 >= delta_LOW(a,b,u));

  lemma_1_6: AXIOM (FORALL (a,b:nonnegreal, u:(fintime?)):
    delta_HIGH(a,b,u) <= (b*dur(u) + U_2*dur(u)*dur(u)/2));

 % The following lemma is commented out because it has the unprovable
 % TCC W <= U_1.  However, it appears not to be needed in any invariant
 % proofs.
 % lemma_1_7: AXIOM delta_HIGH(W - U_1,W,fintime(I)) = W*I - U_1*I*I/2;

  lemma_1_8: AXIOM (FORALL (a,b,c:nonnegreal, t,u:(fintime?)):
    delta_HIGH(a,b,u) + delta_HIGH(b,c,t) <= delta_HIGH(a,c,u+t));

  lemma_1_9: AXIOM (FORALL (a,b:nonnegreal, u:(fintime?)):
    delta_HIGH(a,b,u) >= (a + b)*dur(u)/2);

  lemma_1_10: AXIOM (FORALL (a,b:nonnegreal, u:(fintime?)):
    delta_HIGH(a,b,u) <= (a*dur(u) + U_1*dur(u)*dur(u)/2));

  bool_pred(b:boolean):boolean = true;

%% num_pumps_pred(n_max:num_pumps)(n:{n:num_pumps | n <= n_max}):bool = true;
  num_pumps_pred(n_max:num_pumps)(n:num_pumps):bool = n <= n_max;

  water_level_init_pred(q:water_level):bool = (M_1 < q & q < M_2);
%% water_level_init_pred(q:water_level):bool =
%% % old version before correction -> (M_1 < q & q < M_2);
%%   (M_1 + U_1*I*I/2 < q & q < M_2 - P*(I-S)*num_pumps);

  steam_rate_pred(v_old:nonnegreal,delta_t:(fintime?))
                 (v_new:nonnegreal):bool =
    v_old - U_2*dur(delta_t) <= v_new & v_new <= v_old + U_1*dur(delta_t);

  water_level_pred(q_old:water_level,pr:num_pumps,v_old,v_new:nonnegreal,
```

```
                    delta_t:(fintime?))(q_new:water_level):bool =
      q_old + pr*P*dur(delta_t) - delta_HIGH(v_old,v_new,delta_t)
        <= q_new
    & q_new
        <= q_old + pr*P*dur(delta_t) - delta_LOW(v_old,v_new,delta_t);
  delta_t: VAR (fintime?);
  e_stop: VAR boolean;
  pset,w,p: VAR num_pumps;
  s: VAR nonnegreal;
  actions : DATATYPE
    BEGIN
        nu(timeof:(fintime?)): nu?
        actuator(e_stop_of:boolean, pset_of:num_pumps): actuator?
        sensor(s_of:steam_rate, w_of:water_level, p_of:num_pumps): sensor?
        controller: controller?
        activate: activate?
    END actions;
  a: VAR actions;
  MMTstates: TYPE = [# do_output_part: boolean,
                       stopmode_part: boolean,
                       wl_part: water_level,
                       sr_part: steam_rate,
                       pumps_part: num_pumps,
                       px_part: num_pumps,
  %  boiler below ------------------------------ controller above %
                       pr_part: num_pumps,
                       q_part: water_level,
                       v_part: nonnegreal,
                       pr_new_part: num_pumps,
%%                     error_part: {n:num_pumps | n <= pr_new_part},
                       error_part: num_pumps,
                       do_sensor_part: boolean,
                       set_part: nonnegreal,
                       read_part: nonnegreal,
                       stop_part: boolean #];
  IMPORTING timed_auto_lib@states[actions,MMTstates,time,fintime?]
% Definitions providing simple access to state variables.
  do_output(s:states):boolean = do_output_part(basic(s));
  stopmode(s:states):boolean = stopmode_part(basic(s));
  wl(s:states):water_level = wl_part(basic(s));
  sr(s:states):steam_rate = sr_part(basic(s));
  pumps(s:states):num_pumps = pumps_part(basic(s));
  px(s:states):num_pumps = px_part(basic(s));
  pr(s:states):num_pumps = pr_part(basic(s));
  q(s:states):water_level = q_part(basic(s));
```

```
    v(s:states):nonnegreal = v_part(basic(s));
    pr_new(s:states):num_pumps = pr_new_part(basic(s));
%% error(s:states):{n:num_pumps | n <= pr_new(s)} = error_part(basic(s));
    error(s:states):num_pumps = error_part(basic(s));
    do_sensor(s:states):bool = do_sensor_part(basic(s));
    set(s:states):nonnegreal = set_part(basic(s));
    read(s:states):nonnegreal = read_part(basic(s));
    stop(s:states):bool = stop_part(basic(s));
    OKstate?(s:states):bool = true;
    enabled_general (a:actions, s:states):bool =
         now(s) >= first(s)(a) & now(s) <= last(s)(a);
    enabled_specific (a:actions, s:states):bool =
         CASES a OF
           nu(delta_t): (delta_t > zero
                       & stop(s) = false
                       & now(s) + delta_t <= fintime(read(s))
                       & now(s) + delta_t <= fintime(set(s))),
           actuator(e_stop,pset): (do_output(s) = true
                                 & pset = px(s)
                                 & e_stop = stopmode(s)),
           sensor(steam,w,p): (now(s) = fintime(read(s))
                            & do_sensor(s) = true
                            & stop(s) = false
                            & w = q(s)
                            & steam = v(s)
                            & p = pr(s)),
           activate: (now(s) = fintime(set(s)) & stop(s) = false),
           controller: true
         ENDCASES;
    trans (a:actions, s:states):states =
         CASES a OF
           nu(delta_t): LET new_v_part = epsilon(steam_rate_pred(v(s),delta_t)),
                            new_q_part = epsilon(water_level_pred(q(s),pr(s),
                                                 v(s),new_v_part,delta_t)) IN
                        s WITH [now := now(s) + delta_t,
                                basic := basic(s) WITH
                                  [v_part := new_v_part, q_part := new_q_part]],
           actuator(e_stop,pset): s WITH [basic := basic(s) WITH
                                           [pr_new_part := pset,
                                            stop_part := e_stop,
                                            do_sensor_part := true,
                                            do_output_part := false,
                                            read_part := dur(now(s)) + I]],
           sensor(steam,w,p): s WITH
             [basic := basic(s) WITH
              [sr_part := steam,
               wl_part := w,
               pumps_part := p,
```

```
                do_sensor_part := false,
                do_output_part := true,
                stopmode_part :=
                  IF safety_checks THEN true ELSE epsilon(bool_pred) ENDIF
                  WHERE safety_checks =
%%                  steam <= W - U_1*I OR
                    steam >= W - U_1*I OR
                    w >= M_2 - P*(p*S + max_pumps_after_set*(I - S))
                             + min_steam_water(steam) OR
                    w <= M_1 - P*(p*S + min_pumps_after_set*(I - S))
                             + max_steam_water(steam) ]],
            activate: LET new_error_part = epsilon(num_pumps_pred(pr_new(s))) IN
                      s WITH [basic := basic(s) WITH
                       [set_part := read(s) + S,
                        error_part := new_error_part,
%%                      pr_part := pr_new(s) - error_part]],
                        pr_part := pr_new(s) - new_error_part]],
            controller: s WITH [basic := basic(s) WITH
                                 [px_part := epsilon(num_pumps_pred(num_pumps))]]
        ENDCASES
    enabled (a:actions, s:states):bool =
        enabled_general(a,s) & enabled_specific(a,s) & OKstate?(trans(a,s));
    start (s:states):bool =
        s = LET init_water_level = epsilon(water_level_init_pred) IN
            (# basic := (# do_output_part := false,
%%                         stopmode_part := false,
                           stopmode_part := true,
                           wl_part := init_water_level,
                           sr_part := 0,
                           pumps_part := 0,
                           px_part := 0,
    % boiler part below ---------------------------- controller part above %
                           pr_part := 0,
                           q_part := init_water_level,
                           v_part := 0,
                           pr_new_part := 0,
                           error_part := 0,
                           do_sensor_part := true,
                           set_part := S,
                           read_part := 0,
                           stop_part := false #),
                 now := zero,
                 first := (LAMBDA a: zero),
                 last := (LAMBDA a: infinity) #)
    IMPORTING timed_auto_lib@machine[states, actions, enabled, trans, start]
END boilersys_decls
```

Appendix 4.2 : Local Theories for the Boiler Control System.

```
boilersys_rewrite_aux_1: THEORY
  BEGIN
    IMPORTING boilersys_decls
    nu_rewrite_1: LEMMA (FORALL (t: (fintime?), a: actions):
      nu(t) = a => t = timeof(a));
    nu_rewrite_1r: LEMMA (FORALL (t: (fintime?), a: actions):
      a = nu(t) => timeof(a) = t);
    actuator_rewrite_1_1: LEMMA (FORALL (e: boolean, p: num_pumps, a: actions):
      actuator(e,p) = a => e = e_stop_of(a);
    actuator_rewrite_1_1r: LEMMA (FORALL (e: boolean, p: num_pumps, a: actions):
      a = actuator(e,p) => e_stop_of(a) = e;
    actuator_rewrite_1_2: LEMMA (FORALL (e: boolean, p: num_pumps, a: actions):
      actuator(e,p) = a => p = pset_of(a));
    actuator_rewrite_1_2r: LEMMA (FORALL (e: boolean, p: num_pumps, a: actions):
      a = actuator(e,p) => pset_of(a) = p);
    sensor_rewrite_1_1: LEMMA
     (FORALL (s: steam_rate, w: water_level, p: num_pumps, a: actions):
        sensor(s,w,p) = a => s = s_of(a));
    sensor_rewrite_1_1r: LEMMA
     (FORALL (s: steam_rate, w: water_level, p: num_pumps, a: actions):
        a = sensor(s,w,p) => s_of(a) = s);
    sensor_rewrite_1_2: LEMMA
     (FORALL (s: steam_rate, w: water_level, p: num_pumps, a: actions):
        sensor(s,w,p) = a => w = w_of(a));
    sensor_rewrite_1_2r: LEMMA
     (FORALL (s: steam_rate, w: water_level, p: num_pumps, a: actions):
        a = sensor(s,w,p) => w_of(a) = w);
    sensor_rewrite_1_3: LEMMA
     (FORALL (s: steam_rate, w: water_level, p: num_pumps, a: actions):
        sensor(s,w,p) = a => p = p_of(a));
    sensor_rewrite_1_3r: LEMMA
     (FORALL (s: steam_rate, w: water_level, p: num_pumps, a: actions):
        a = sensor(s,w,p) => p_of(a) = p);
  END boilersys_rewrite_aux_1

boilersys_rewrite_aux_2: THEORY
  BEGIN
    IMPORTING boilersys_decls
    nu_rewrite_2: LEMMA (FORALL (t: (fintime?), a: actions):
      nu?(a) & t = timeof(a) => nu(t) = a);
    nu_rewrite_3: LEMMA (FORALL (t: (fintime?), a: actions):
      nu?(a) => nu(timeof(a)) = a);
    actuator_rewrite_2: LEMMA (FORALL (e: boolean, p: num_pumps, a: actions):
      (actuator?(a) & e = e_stop_of(a) & p = pset_of(a)) => actuator(e,p) = a);
```

```
    actuator_rewrite_3: LEMMA (FORALL (e: boolean, p: num_pumps, a: actions):
      actuator?(a) => actuator(e_stop_of(a),pset_of(a)) = a);
    sensor_rewrite_2: LEMMA
      (FORALL (s: steam_rate, w: water_level, p: num_pumps, a: actions):
        (sensor?(a) & s=s_of(a) & w=w_of(a) & p=p_of(a)) => sensor(s,w,p) = a);
    sensor_rewrite_3: LEMMA
      (FORALL (s: steam_rate, w: water_level, p: num_pumps, a: actions):
        sensor?(a) => sensor(s_of(a), w_of(a), p_of(a)) = a);
  END boilersys_rewrite_aux_2

boilersys_unique_aux: THEORY
  BEGIN
    IMPORTING boilersys_decls
    nu_unique: LEMMA (FORALL (t1, t2: (fintime?)): (nu(t1)=nu(t2) => t1 = t2));
    actuator_unique: LEMMA (FORALL (e1,e2: boolean, p1,p2: num_pumps):
      (actuator(e1,p1) = actuator(e2,p2) => e1 = e2 & p1 = p2));
    sensor_unique: LEMMA
      (FORALL (s1,s2: steam_rate, w1,w2: water_level, p1,p2: num_pumps):
        (sensor(s1,w1,p1) = sensor(s2,w2,p2) => s1=s2 & w1=w2 & p1=p2));
  END boilersys_unique_aux
```

Appendix 4.3 : Local Strategies for the Boiler Control System.

```
(defstep |boilersys_simp| ()
  (then  (expand "do_output")
         (expand "stopmode")
         (expand "wl")
         (expand "sr")
         (expand "pumps")
         (expand "px")
         (expand "max_pumps_after_set")
         (expand "min_pumps_after_set")
         (expand "max_steam_water")
         (expand "min_steam_water")
         (expand "pr")
         (expand "q")
         (expand "v")
         (expand "pr_new")
         (expand "error")
         (expand "do_sensor")
         (expand "set")
         (expand "read")
         (expand "stop")
         (flatten))
  ""
  "Expanding some boilersys definitions")
```